Table of Contents

Introduction 1.1

x86 Course 1.2

- Part 1: Goals 1.2.1
- Part 2: Techniques 1.2.2
- Part 3: Types Of Malware 1.2.3
- Part 4: x86 Assembly Intro 1.2.4
- Part 5: Binary Number System 1.2.5
- Part 6: Hexadecimal Number System 1.2.6
- Part 7: Transistors And Memory 1.2.7
- Part 8 - Bytes, Words, Double Words, etc... 1.2.8
- Part 9: x86 Basic Architecture 1.2.9
- Part 10: General-purpose Registers 1.2.10
- Part 11: Segment Registers 1.2.11
- Part 12: Instruction Pointer Register 1.2.12
- Part 13: Control Registers 1.2.13
- Part 14: Flags 1.2.14
- Part 15: Stack 1.2.15
- Part 16: Heap 1.2.16
- Part 17 – How To Install Linux 1.2.17
- Part 18 - vim Text Editor 1.2.18
- Part 19 - Why Learn Assembly 1.2.19
- Part 20 - Instruction Code Handling 1.2.20
- Part 21 - How To Compile A Program 1.2.21
- Part 22 - ASM Program 1 [Moving Immediate Data] 1.2.22
- Part 23 - ASM Debugging 1 [Moving Immediate Data] 1.2.23
- Part 24 - ASM Hacking 1 [Moving Immediate Data] 1.2.24
- Part 25 - ASM Program 2 [Moving Data Between Registers] 1.2.25
- Part 26 - ASM Debugging 2 [Moving Data Between Registers] 1.2.26
- Part 27 - ASM Hacking 2 [Moving Data Between Registers] 1.2.27
- Part 28 - ASM Program 3 [Moving Data Between Memory And Registers] 1.2.28
- Part 29 - ASM Debugging 3 [Moving Data Between Memory And Registers] 1.2.29
- Part 30 - ASM Hacking 3 [Moving Data Between Memory And Registers] 1.2.30

Part 31 - ASM Program 4 [Moving Data Between Registers And Memory] 1.2.31
Part 32 - ASM Debugging 4 [Moving Data Between Registers And Memory] 1.2.32
Part 33 - ASM Hacking 4 [Moving Data Between Registers And Memory] 1.2.33
Part 34 - ASM Program 5 [Indirect Addressing With Registers] 1.2.34
Part 35 - ASM Debugging 5 [Indirect Addressing With Registers] 1.2.35
Part 36 - ASM Hacking 5 [Indirect Addressing With Registers] 1.2.36
Part 37 - ASM Program 6 [CMOV Instructions] 1.2.37
Part 38 - ASM Debugging 6 [CMOV Instructions] 1.2.38
Part 39 - ASM Hacking 6 [CMOV Instructions] 1.2.39
Part 40 - Conclusion 1.2.40
ARM-32 Course 1 1.3
Part 1 – The Meaning Of Life 1.3.1
Part 2 - Number Systems 1.3.2
Part 3 - Binary Addition 1.3.3
Part 4 - Binary Subtraction 1.3.4
Part 5 - Word Lengths 1.3.5
Part 6 - Registers 1.3.6
Part 7 - Program Counter 1.3.7
Part 8 - CPSR 1.3.8
Part 9 - Link Register 1.3.9
Part 10 - Stack Pointer 1.3.10
Part 11 - ARM Firmware Boot Procedures 1.3.11
Part 12 - Von Neumann Architecture 1.3.12
Part 13 - Instruction Pipeline 1.3.13
Part 14 - ADD 1.3.14
Part 15 - Debugging ADD 1.3.15
Part 16 - Hacking ADD 1.3.16
Part 17 - ADDS 1.3.17
Part 18 – Debugging ADDS 1.3.18
Part 19 – Hacking ADDS 1.3.19
Part 20 – ADC 1.3.20
Part 21 – Debugging ADC 1.3.21
Part 22 – Hacking ADC 1.3.22
Part 23 – SUB 1.3.23
Part 24 – Debugging SUB 1.3.24
Part 25 – Hacking SUB 1.3.25

ARM-32 Course 2 1.4
Part 1 – The Meaning Of Life Part 2 1.4.1
Part 2 – Number Systems 1.4.2
Part 3 – Binary Addition 1.4.3
Part 4 – Binary Subtraction 1.4.4
Part 5 – Word Lengths 1.4.5
Part 6 – Registers 1.4.6
Part 7 – Program Counter 1.4.7
Part 8 - CPSR 1.4.8
Part 9 - Link Register 1.4.9
Part 10 - Stack Pointer 1.4.10
Part 11 - Firmware Boot Procedures 1.4.11
Part 12 - Von Neumann Architecture 1.4.12
Part 13 - Instruction Pipeline 1.4.13
Part 14 - Hello World 1.4.14
Part 15 - Debugging Hello World 1.4.15
Part 16 - Hacking Hello World 1.4.16
Part 17 - Constants 1.4.17
Part 18 – Debugging Constants 1.4.18
Part 19 – Hacking Constants 1.4.19
Part 20 – Character Variables 1.4.20
Part 21 – Debugging Character Variables 1.4.21
Part 22 – Hacking Character Variables 1.4.22
Part 23 – Boolean Variables 1.4.23
Part 24 – Debugging Boolean Variables 1.4.24
Part 25 – Hacking Boolean Variables 1.4.25
Part 26 – Integer Variables 1.4.26
Part 27 – Debugging Integer Variables 1.4.27
Part 28 – Hacking Integer Variables 1.4.28
Part 29 – Float Variables 1.4.29
Part 30 – Debugging Float Variables 1.4.30
Part 31 – Hacking Float Variables 1.4.31
Part 32 – Double Variables 1.4.32
Part 33 – Debugging Double Variables 1.4.33
Part 34 – Hacking Double Variables 1.4.34
Part 35 – SizeOf Operator 1.4.35
Part 36 – Debugging SizeOf Operator 1.4.36
Part 37 – Hacking SizeOf Operator 1.4.37

Part 38 – Pre-Increment Operator 1.4.38
Part 39 – Debugging Pre-Increment Operator 1.4.39
Part 40 – Hacking Pre-Increment Operator 1.4.40
Part 41 – Post-Increment Operator 1.4.41
Part 42 – Debugging Post-Increment Operator 1.4.42
Part 43 – Hacking Post-Increment Operator 1.4.43
Part 44 – Pre-Decrement Operator 1.4.44
Part 45 – Debugging Pre-Decrement Operator 1.4.45
Part 46 – Hacking Pre-Decrement Operator 1.4.46
Part 47 – Post-Decrement Operator 1.4.47
Part 48 – Debugging Post-Decrement Operator 1.4.48
Part 49 – Hacking Post-Decrement Operator 1.4.49
x64 Course 1.5
Part 1 – The Cyber Revolution 1.5.1
Part 2 - Transistors 1.5.2
Part 3 - Logic Gates 1.5.3
Part 4 - Number Systems 1.5.4
Part 5 - Binary Addition 1.5.5
Part 6 - Binary Subtraction 1.5.6
Part 7 - Word Lengths 1.5.7
Part 8 - General Architecture 1.5.8
Part 9 - Calling Conventions 1.5.9
Part 10 - Boolean Instructions 1.5.10
Part 11 - Pointers 1.5.11
Part 12 - Load Effective Address 1.5.12
Part 13 - The Data Segment 1.5.13
Part 14 - SHL Instruction 1.5.14
Part 15 - SHR Instruction 1.5.15
Part 16 - ROL Instruction 1.5.16
Part 17 - ROR Instruction 1.5.17
Part 18 - Boot Sector Basics [Part 1] 1.5.18
Part 19 - Boot Sector Basics [Part 2] 1.5.19
Part 20 - Boot Sector Basics [Part 3] 1.5.20
Part 21 - Boot Sector Basics [Part 4] 1.5.21
Part 22 - Boot Sector Basics [Part 5] 1.5.22
Part 23 - Boot Sector Basics [Part 6] 1.5.23
Part 24 - Boot Sector Basics [Part 7] 1.5.24
Part 25 - Boot Sector Basics [Part 8] 1.5.25

- Part 26 - Boot Sector Basics [Part 9] 1.5.26
- Part 27 - x64 Assembly [Part 1] 1.5.27
- Part 28 - x64 Assembly [Part 2] 1.5.28
- Part 29 - x64 Assembly [Part 3] 1.5.29
- Part 30 - x64 Assembly [Part 4] 1.5.30
- Part 31 - x64 Assembly [Part 5] 1.5.31
- Part 32 - x64 Assembly [Part 6] 1.5.32
- Part 33 - x64 Assembly [Part 7] 1.5.33
- Part 34 - x64 C++ 1 Code [Part 1] 1.5.34
- Part 35 - x64 C++ 2 Debug [Part 2] 1.5.35
- Part 36 - x64 C++ 3 Hacking [Part 3] 1.5.36
- Part 37 - x64 C & Genesis Of Life 1.5.37
- Part 38 - x64 Networking Basics 1.5.38
- Part 39 - Why C? 1.5.39
- Part 40 - Hacking Hello World! 1.5.40
- Part 41 - Hacking Variables! 1.5.41
- Part 42 - Hacking Branches! 1.5.42
- Part 43 - Hacking Pointers! 1.5.43

ARM-64 Course 1.6

- Part 1 - The Meaning Of Life 1.6.1
- Part 2 - Development Setup 1.6.2
- Part 3 - "Hello World" 1.6.3
- Part 4 - Debugging "Hello World" 1.6.4
- Part 5 - Hacking "Hello World" 1.6.5
- Part 6 - Basic I/O 1.6.6
- Part 7 - Debugging Basic I/O 1.6.7
- Part 8 - Hacking Basic I/O 1.6.8
- Part 9 - Character Primitive Datatype 1.6.9
- Part 10 - Debugging Character Primitive Datatype 1.6.10
- Part 11 - Hacking Character Primitive Datatype 1.6.11
- Part 12 - Boolean Primitive Datatype 1.6.12
- Part 13 - Debugging Boolean Primitive Datatype 1.6.13
- Part 14 - Hacking Boolean Primitive Datatype 1.6.14
- Part 15 - Float Primitive Datatype 1.6.15
- Part 16 - Debugging Float Primitive Datatype 1.6.16
- Part 17 - Hacking Float Primitive Datatype 1.6.17
- Part 18 - Double Primitive Datatype 1.6.18
- Part 19 - Debugging Double Primitive Datatype 1.6.19

Part 20 - Hacking Double Primitive Datatype 1.6.20
Pico Hacking Course 1.7
Part 1 - The Why, The How... 1.7.1
Part 2 - Hello World 1.7.2
Part 3 - Debugging Hello World 1.7.3
Part 4 - Hacking Hello World 1.7.4
Part 5 - char 1.7.5
Part 6 - Debugging char 1.7.6
Part 7 - Hacking char 1.7.7
Part 8 - int 1.7.8
Part 9 - Debugging int 1.7.9
Part 10 - Hacking int 1.7.10
Part 11 - float 1.7.11
Part 12 - Debugging float 1.7.12
Part 13 - Hacking float 1.7.13
Part 14 - double 1.7.14
Part 15 - Debugging double 1.7.15
Part 16 - Hacking double 1.7.16
Part 17 - "ABSOLUTE POWER CORRUPTS ABSOLUTELY!", The Tragic Tale Of Input... 1.7.17
Part 18 - "FOR 800 YEARS HAVE I TRAINED JEDI!", The FORCE That IS Input... 1.7.18
Part 19 - Input 1.7.19
Part 20 - Debugging Input 1.7.20

Reverse Engineering For Everyone!

— by @mytechnotalent

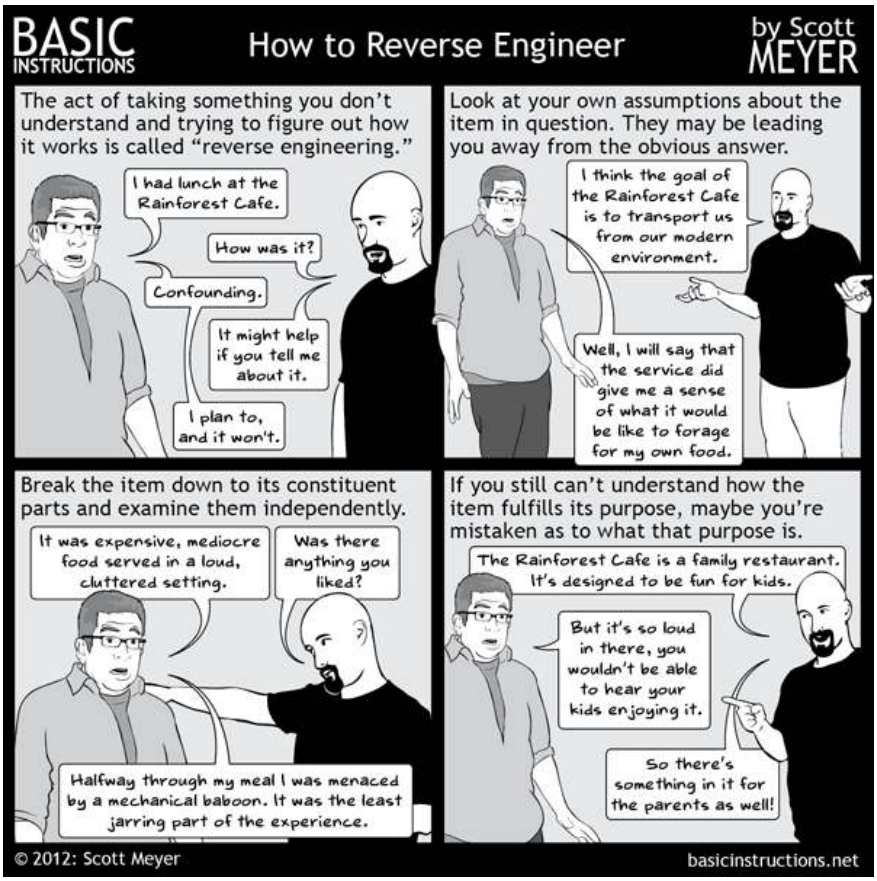


Wait, what's reverse engineering?

Wikipedia defines it as:

> Reverse engineering, also called backwards engineering or back engineering, is the process by which an artificial object is deconstructed to reveal its designs, architecture, code, or to extract knowledge from the object. It is similar to scientific research, the only difference being that scientific research is conducted into a natural phenomenon.

Whew, that was quite a mouthful, wasn't it? Well, it is one of the main reasons why this tutorial set exists. To make reverse engineering *as simple as possible.*



This comprehensive set of reverse engineering tutorials covers x86, x64 as well as 32-bit ARM and 64-bit architectures. If you're a newbie looking to learn reversing, or just someone looking to revise on some concepts, you're at the right place. As a beginner, these tutorials will carry you from nothing upto the mid-basics of reverse engineering, a skill that everyone within the realm of cyber-

security should possess. If you're here just to refresh some concepts, you can conveniently use the side bar to take a look at the sections that has been covered so far.

You can get the entire tutorial set in PDF or MOBI format. All these ebook versions will get updated automatically as new tutorials will be added.

Download here: [PDF | MOBI]

Gitbook crafted with ♡ by @0xInfection

The x86 Architecture

Let's dive in rightaway!

Part 1: Goals

Essential to the discussion of basic reverse engineering is the concept of modern malware analysis. Malware analysis is the understanding and examination of information necessary to respond to a network intrusion.

This short tutorial will begin with the basic concepts of malware reverse engineering and graduate to an entry-level basic examination of Assembly Language.

The keys to the kingdom so to speak are rooted in the break-down of the respective suspected malware binary and how to find it on your network and ultimately to contain it.

Upon full identification of the files required for deeper analysis, it is critical to develop signatures to detect malware infections throughout your network whether it be a home based LAN or complex corporate WAN to which malware analysis is necessary to develop host-based and network signatures.

To begin with the concept of a host-based signature, we need to understand that these are utilized to find malicious code in a target machine. Host-based signatures are also referred to as indicators which can identify files created or edited by the infected code which can make hidden changes to a computers registry. This is quite in contrast with antivirus signatures because these concentrate on what the malware actually does rather than the make-up of the malware which makes them more effective in finding malware that can migrate or has been removed from the media.

In contrast, network signatures are used to find malicious code by examining network traffic. It is important to note such tools as WireShark and the like are often effective in such analysis.

Upon identification of these aforementioned signatures, the next step is to identify what the malware is actually doing.

In our next lesson we will discuss techniques of malware analysis.

Part 2: Techniques

There are two basic techniques that you can employ when analyzing malware. The first being static analysis and the other being dynamic analysis.

Static analysis uses software tools to examine the executable without running the actual decompiled instructions in Assembly. We will not focus on this type of analysis here as we are going to focus on actual disassembled binaries instead however in future courses we will.

Dynamic analysis uses disassemblers and debuggers to analyze malware binaries while actually running them. The most popular tool in the market today is called IDA which is a multi-platform, multi-processor disassembler and debugger. There are other disassembler/debugger tools as well on the market today such as Hopper Disassembler, OllyDbg and many more.

A disassembler will convert an executable binary written in Assembly, C, C++, etc into Assembly Language instructions that you can debug and manipulate.

Reverse engineering is much more than just malware analysis. At the end of our series, our capstone tutorial will utilize IDA as we will create a real-world scenario where you will be tasked by the CEO of ABC Biochemicals to secretly try to ethically hack his companies software that controls a bullet-proof door in a very sensitive Bio-Chemical lab in order to test how well the software works against real threats. The project will be very basic however it will ultimately showcase the power of Assembly Language and how one can use it to reverse engineer and ultimately provide solutions on how to better design the code to make it safer.

In our next lesson we will discuss various types of malware.

Part 3: Types Of Malware

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Malware falls into several categories of which I will touch briefly upon below.

A backdoor is malicious code that embeds itself into a computer to allow a remote attacker access with very little or sometimes no authority to execute various commands on any respective local computer.

A botnet allows an attacker access to a system however receive instructions not from one remote attacker but from a command-and-control server to which can control an unlimited amount of computers at the same time.

A downloader is nothing more than malicious code that has only one purpose which is to install other malicious software. Downloaders are frequently installed when a hacker gains access to a system initially. The downloader then installs additional software to control the system.

We find information access malware which gathers information from a computer and sends it directly to a host such as a keylogger or password grabber and usually used to obtain access to various online accounts that can be very sensitive.

There are malicious programs that launch other malicious programs which use non-standard options to get increased access or a greater cloaking/hiding technique when penetrating a system.

One of the most dangerous forms of malware is the rootkit which hides the existence of itself and additional malware from the user which makes it extremely hard to locate. A rootkit can manipulate processes such as hiding their IP in an IP scan so that a user may never know that they have a direct socket to a botnet or other remote computer.

Scareware is used to trick a user into purchasing additional software to falsely protect a user when there is no real threat whatsoever that exists. Once a user pays to have the tricked software removed from the computer it then can stay resident and later emerge in an altered form.

There are also various kinds of malware that send spam from a target machine which generates income for the attacker by allowing them to sell various services to other users.

The final form of malware is that of a traditional worm or virus which copies itself and goes after other computers.

This is the end the road for now regarding our discussion of malware because we first need to go back to the beginning and understand how a computer works at it's base level.

In our next lesson we will begin our long journey into x86 Assembly Language. In order to truly understand the very basics of reverse engineering and malware we need to over the next several months take a deep dive into the core and build our way up.

Part 4: x86 Assembly Intro

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Ladies and Gentlemen, boys and girls, children of all ages! We are about to embark on a journey that will change your life forever!

There is vast material to cover to get a good understanding of Assembly Language and why it is important to understand the basics.

The first question we must answer is what is x86 Assembly Language to which the answer is a family of backward-compatible Assembly Languages which provide compatibility back to the Intel 8000 series of microprocessors. x86 Assembly Languages are used to produce object code for the aforementioned series of processors. It uses mnemonics to represent the instructions that the CPU can execute.

Assembly Language for the x86 microprocessor works in conjunction with various operating systems. We will focus on Linux Assembly Language utilizing the Intel syntax in addition to learning how to program in C to which we will disassemble the source code an analyze the respective Assembly.

x86 Assembly Language has two choices of syntax. The AT&T syntax was dominant in the Unix world since the OS was developed at AT&T Bell Labs. In contrast, the Intel syntax was originally used for the documentation of the x86 platform and was dominant in the MS-DOS and Windows environments.

For our purposes, when we are ultimately disassembling or debugging software, whether it be in a Linux or Windows environment, we will see the Intel syntax in large measure. This is essential whether we are examining a Windows binary in PE format or a Linux binary in ELF format. More on that later in this tutorial.

The main differences between the two is in the AT&T syntax, the source comes before the destination and in the Intel syntax, the destination comes before the source. We will discuss this in more detail later in the tutorial.

Before you run for the door and regret embarking on this journey, remember, some basic context helps to which we will develop throughout our quest. Many of these topics may be confusing at this point which is perfectly normal as we will develop them in time.

We will focus on Linux Assembly because Linux runs on a variety of hardware and is capable of running devices such as a cell phone, personal computer or a complex commercial server.

Linux is also open source and there are many versions. We will focus on Ubuntu in our demonstrations which can be freely obtained. In contrast, the Windows operating system is owned and controlled by one company, Microsoft, to which all updates, security patches and service patches come directly from them where Linux has millions of professionals providing the same absolutely free!

We will also focus on a 32-bit architecture as ultimately most malware will be written for such in order to infect as many systems as possible. 32-bit applications/malware will work on 64-bit systems so we want to understand the basics of the 32-bit world.

In our next lesson we discuss the binary number system. Grab your cup of coffee you are going to need it!

Part 5: Binary Number System

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Binary numbers are what define the core of a computer. A bit within a computer is either on or off. A bit has either electricity turned on to it or it is absent of such. We will dive into this deeper in future tutorials.

Puzzled and confused, where do we go from here?

Have no fear! The binary number system is here! It is important to understand that in binary, each column has a value two times the column to its right and there are only two digits in the base which happen to be 0 and 1.

In decimal, base 10, say we have the number 15 which means (1 x 10) + (5 x 1) = 15 therefore the 5 is the number times 1 and the 1 is that number times 10.

Binary works in a similar fashion however we are now referring to base 2. That same number in binary is 1111. To illustrate:

```
1   1   1   1
8s  4s  2s  1s

(8 x 1) + (4 x 1) + (2 x 1) + (1 x 1)
   8    +    4    +    2    +    1    = 15
```

Binary numbers are important because using them instead of the decimal system simplifies the design of computers and related technologies. The simplest definition of the binary number system is a system of numbering that uses only two digits, as we mentioned above, to represent numbers necessary for a computer architecture rather than using the digits 1 through 9 plus 0 to represent such.

In our next lesson we discuss the hexadecimal number system. It only gets more exciting from here!

Part 6: Hexadecimal Number System

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Now that we are binary masters, it's time to tackle the numbering system of numbering systems!

We learned in binary that each number represents a bit. If we combine 8 bits, we get a byte. A byte can be further subdivided into its top 4 bits and its low 4 bits. A combination of 4 bits is a nibble. Since 4 bits gives you the possible range from 0 - 15 a base 16 number system is easier to work with. Keep in mind when we say base 16 we start with 0 and therefore 0 - 15 is 16 different numbers.

This exciting number system is called hexadecimal. The reason why we use this number system is that in x86 Assembly it is much easier to express binary number representations in hexadecimal than it is in any other numbering system.

Hexadecimal is similar to every other number system except in hexadecimal, each column has a value of 16 times the value of the column to its right. The fun part about hexadecimal is that not only do we have 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 we have A, B, C, D, E and F and therefore 16 different symbols.

Lets look at a simple table to see how hexadecimal compares to decimal.

Decimal	Hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Ok I see the smoke coming out of your ears but its ok! In decimal, everything is dealt with in the power of 10. Let's take the number 42 and examine it in decimal:

2 x 10 ^ 0 = 2

4 x 10 ^ 1 = 40

Remember 10 to the 0 power is 1 and 10 to the 1st power is 10, therefore, 2 + 40 = 42.

Grab your coffee, here comes the fun stuff!

If we understand that decimal is a base 10 number system, we can create a simple formula where b represents the base. In this case, b = 10.

(2 * b ^ 0) + (4 * b ^ 1)

(2 * 10 ^ 0) + (4 * 10 ^ 1) = 42

In binary, 42 decimal is 0010 1010 binary as follows:

0 x 2 ^ 0 = 0

1 x 2 ^ 1 = 2

0 x 2 ^ 2 = 0

1 x 2 ^ 3 = 8

0 x 2 ^ 4 = 0

1 x 2 ^ 5 = 32

0 x 2 ^ 6 = 0

0 x 2 ^ 7 = 0

0 + 2 + 0 + 8 + 0 + 32 + 0 + 0 = 42 decimal

In hexadecimal, everything is dealt with in the power of 16. Therefore 42 in decimal is 2A in hexadecimal:

10 * 16 ^ 0 = 10

2 * 16 ^ 1 = 32

10 + 32 = 42 decimal => 2A hexadecimal

This is the same as saying:

10 * 1 = 10

2 * 16 = 32

10 + 32 = 42 decimal => 2A hexadecimal

Keep in mind 10 decimal is equal to A hexadecimal and 2 decimal is equal to 2 hexadecimal. In our formula above when we deal with A, B, C, D, E or F we need to convert them to their decimal equivalent.

Lets take another example of F5 hexadecimal. This would be as follows:

5 x 16 ^ 0 = 5

15 x 16 ^ 1 = 240

5 + 240 = 245 decimal => F5 hexadecimal

Lets look at a binary to hexadecimal table:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

It is important to understand that every hexadecimal number is 4 bits long or called a nibble. This will become critical when we are reverse engineering our C programs into Assembly.

Lets look at this another way. Lets work with some more hexadecimal numbers and convert them to decimal:

Hexadecimal	Decimal
3A	(3 x 16) + (10 x 1) = 58
F1	(15 x 16) + (1 x 1) = 241
4AB	(4 x 256) + (10 x 16) + (11 x 1) = 1,195
F1CD	(15 x 4096) + (1 x 256) + (12 x 16) + (13 x 1) = 61,901

To re-emphasize F1CD as a simple conversion:

D --- 13 x 1 = 13

C --- 12 x 16 = 192

1 --- 1 x 256 = 256

F --- 15 x 4096 = 61,440

13 + 192 + 256 + 61,440 = 61,901

Addition in hexadecimal works as follows. From this point forward all numbers in hexadecimal will have a 'h' next to the number:

```
Add

    1 1
  F 0 B A h
+ E 9 A D h
-----------
1 D A 6 7 h

    A + D = 10 + 13
          = 23
          = 16 + 7
          = 17h

1 + B + A = 1 + 11 + 10
          = 22
          = 16 + 6
          = 16h

1 + 0 + 9 = 1 + 0 + 9
          = 10
          = Ah

    F + E = 15 + 14
          = 29
          = 16 + 13
          = 1Dh
```

Another example is as such:

```
Add


  5 C 8 4 h
+   3 2 9 h
-----------
  5 F A D h

    9 + 4 = 13
          = Dh

    8 + 2 = 10
          = Ah

    C + 3 = 12 + 3
          = 15
          = Fh
```

A final add example is as such:

```
Add

      1
  2 A 1 8 h
+   4 3 B h
-----------
  2 E 5 3 h

    8 + B = 13 [1 represents 1 group of 16 with 3 left over.]
1 + 1 + 3 = 5
    A + 4 = E
```

We will now focus on subtraction:

```
Subtract

      7 13
    A 8 3 h
+   4 3 B h
-----------
    6 4 8 h

    3 - B = undefined [B represents 11 in decimal.]
                      [We can't sub 3 from 11.]
                      [We borrow 1 from 8 and make it a 7.]
                      [3 means we have 1 complete group of 16.]
                      [When added to 3 extra equals 19.]
                      [19 - B or 19 - 11 = 8]
    7 - 3 = 4
    A - 4 = 6
```

You are probably asking yourself why is this guy spending so much time going over so many different ways of learning this! The answer is that each of us learn a little different from the next. I wanted to show several representations of hexadecimal compared to decimal and binary to help put together the whole picture.

It is fundamental that you understand what is going on here in order to proceed any further. If you have any questions, please comment below and I will be more than happy to help!

In our next lesson we discuss switches, transistors and memory.

Part 7: Transistors And Memory

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our last lesson, we took a very deep dive into the hexadecimal number system. I am going to keep this weeks lesson short so that you can re-read last weeks lesson. I can not emphasize how important it is to understand hexadecimal number conversions in addition to the ability to manually add and subtract them.

In the real world, we have calculators, in the real world we use the Windows operating system, in the real world professional reverse engineers use GUI debuggers like IDA Pro and others.

The question is, why am I not jumping right into the core of what real reverse engineers do? The answer is simple, one must have a deep respect and understanding of the machine in order to become great. We will never change the world without fully understanding it first. Patience and perseverance win the day.

I focus on on Linux and console-based programming because most professional servers utilize Linux and therefore is the greatest threat of malware. Understanding Linux Assembly allows you to very easily grasp the library-choking portable executable format of Windows Assembly in a much deeper way.

As I step off the soap box, lets get back to the basics of computers so here we go!

When we ask ourselves what is a computer one must go down to as about as basic as one can get.

Electronic computers are simply made out of transistor switches. Transistors are microscopic crystals of silicon that use electrical properties of silicon to act as switches. Modern computers have what are referred to as field-effect transistors.

Let's use an example of 3 pins. When an electrical voltage is applied to pin 1, current then flows between pins 2 and 3. When the voltage is removed from the first pin, current stops flowing between pins 2 and 3.

When we zoom out a bit we see that there are also diodes and capacitors when taken together with the transistor switches we now have a memory cell. A memory cell keeps a minimum current flow to which when you put a small voltage on its input pin and a similar voltage on its select pin, a voltage will appear and remain on its output pin. The output voltage remains in its set state until the voltage is removed from the input pin in conjunction with the select pin.

Why is this important you ask. Very simply, the presence of voltage indicates a binary 1 and the absence of voltage indicates a binary 0 therefore the memory cell holds one binary digit or bit which is either 1 or 0 meaning on or off.

In our next lesson we will discuss bytes and words.

Part 8 - Bytes, Words, Double Words, etc...

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Memory is measured in bytes. A byte is 8 bits. Two bytes are called a word and two words are called a double word which is four bytes (32-bit) and a quad word is eight bytes (64-bit).

A byte is 8 bits and is 2^8 power which is 256. The number of binary numbers 8 bits in size is one of 256 values starting at 0 and going to 255.

Every byte of memory in a computer has its own unique address. Let's review the disassembled instructions for a simple hello world application in Linux by setting a breakpoint on the main function. We will use the GDB debugger:

```
Starting program: /home/noroot/Desktop/Code/Example1/Example1

Breakpoint 1, main () at Example1.c:4
4               printf("hello world");
(gdb) disas main
Dump of assembler code for function main:
   0x0804840b <+0>:     lea    ecx,[esp+0x4]
   0x0804840f <+4>:     and    esp,0xfffffff0
   0x08048412 <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>:    push   ebp
   0x08048416 <+11>:    mov    ebp,esp
   0x08048418 <+13>:    push   ecx
   0x08048419 <+14>:    sub    esp,0x4
=> 0x0804841c <+17>:    sub    esp,0xc
   0x0804841f <+20>:    push   0x80484c0
   0x08048424 <+25>:    call   0x80482e0 <printf@plt>
   0x08048429 <+30>:    add    esp,0x10
   0x0804842c <+33>:    mov    eax,0x0
   0x08048431 <+38>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>:    leave
   0x08048435 <+42>:    lea    esp,[ecx-0x4]
   0x08048438 <+45>:    ret
End of assembler dump.
(gdb)
```

Don't worry if this does not make sense yet. The point of utilizing this example is to give you a sneak peek into our first program that we will examine in addition to learning about memory in a computer.

Below is an examination of the ESP register. Again, it is not critical that you understand what a register is or what ESP does. We simply want to see what a memory location looks like:

```
(gdb) x/1xw $esp
0xffffd040:     0xf7fac3dc
```

We see the memory location of 0xffffd040 which of course is in hexadecimal. We also see the value inside the ESP register which is 0xf7fac3dc which is also in hexadecimal.

It is important to understand that 0xffffd040 is 4 bytes and is a double word. As we learned in Part 6: Hexadecimal Number System, each hexadecimal digit is 4 bits long otherwise called a nibble. In 0xffffd040, lets look at the right most digit of 0. In this example, 0 (hexadecimal) is 4 bits long. If we look at 40 (in hexadecimal), we see that is a byte in length or 8 bits long. If we look at d040, we have two bytes or a word in length. Finally, ffffd040 is a double word or 4 bytes in length which is 32-bits long. The 0x at the beginning of the address just designates that is is a hexadecimal value.

A computer program is nothing more than machine instructions stored in memory. A 32-bit CPU fetches a double word from a memory address. A double word is 4 bytes in a row which is read from memory and loaded into the CPU. As soon as it finishes executing, the CPU fetches the next machine instruction in memory from the instruction pointer.

Those of you new to assembly have now had your first look. Don't get discouraged or frustrated if you do not know what is going on here. We will take our time and go through dozens of examples to break down each step in future lessons. What is important is that you take your time and examine what each lesson is discussing. Please always feel free to comment below with any questions.

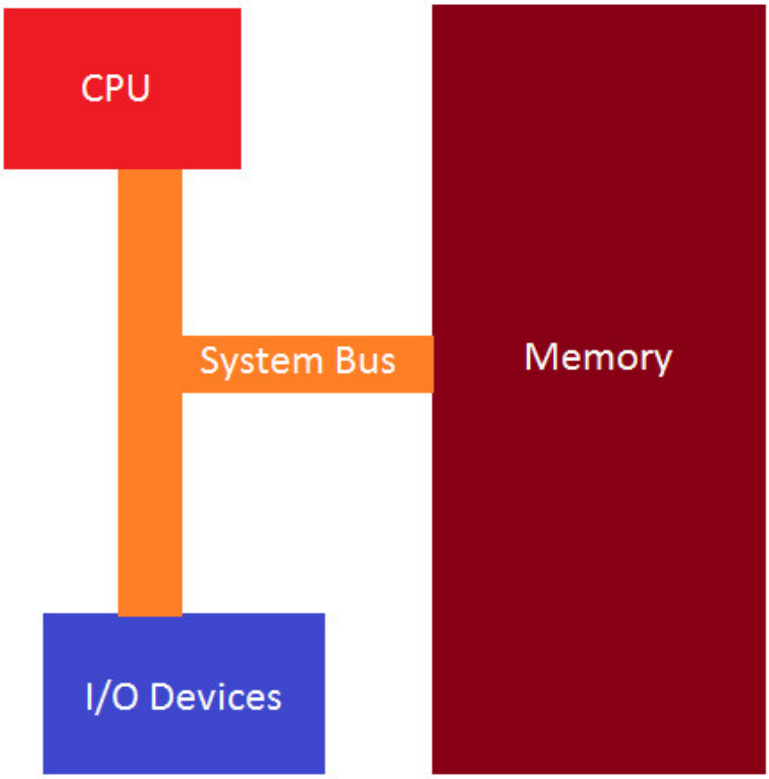
In our next tutorial we will discuss the basics of x86 Architecture.

Part 9: x86 Basic Architecture

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

A computer application is simply a table of machine instructions stored in memory to which the binary numbers which make up the program are unique only in the way the CPU deals with them.

The basic architecture is made up of a CPU, memory and I/O devices which are input/output devices which are all connected by a system bus as detailed below.



The CPU consists of 4 parts which are:

1)Control Unit - Retrieves and decodes instructions from the CPU and then storing and retrieving them to and from memory.

2)Execution Unit - Where the execution of fetching and retrieving instructions occurs.

3)Registers - Internal CPU memory locations used a temporary data storage.

4)Flags - Indicate events when execution occurs.



We will discuss 32-bit x86 so therefore a 32-bit CPU first fetches a double word (4 bytes or 32-bits in length) from a specific address in memory and is read from memory and loaded into the CPU. At this point the CPU looks at the binary pattern of bits within the double word and begins executing the procedure that the fetched machine instruction directs it to do.

Upon completion of executing an instruction, the CPU goes to memory and fetches the next machine instruction in sequence. The CPU has a register, which we will discuss registers in a future tutorial, called the EIP or instruction pointer that contains the address of the next instruction to be fetched from memory and then executed.

We can immediately see that if we controlled flow of EIP, we can alter the program to do things it was NOT intended to do. This is a popular technique upon which malware operates.

The entire fetch and execute process is tied to the system clock which is an oscillator that emits square-wave pulses at precise intervals.

In our next tutorial we will dive deeper into the IA-32 Architecture with a discussion of the General-purpose Registers.

Part 10: General-purpose Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The general-purpose registers are used to temporarily store data as it is processed on the processor. The registers have evolved dramatically over time and continue to do so. We will focus on 32-bit x86 architecture for our purposes.

Each new version of general-purpose registers is created to be backward compatible with previous processors. This means that code utilizing 8-bit registers on the 8080 chips will still function on today's 64-bit chipset.

General-purpose registers can be used to hold any type of data to which some have acquired specific use which are used in programs. Lets review the 8 general-purpose registers in an IA-32 architecture.

EAX: Main register used in arithmetic calculations. Also known as accumulator, as it holds results of arithmetic operations and function return values.

EBX: The Base Register. Pointer to data in the DS segment. Used to store the base address of the program.

ECX: The Counter register is often used to hold a value representing the number of times a process is to be repeated. Used for loop and string operations.

EDX: A general purpose register. Additionally used for I/O operations. In addition will extend EAX to 64-bits.

ESI: Source Index register. Pointer to data in the segment pointed to by the DS register. Used as an offset address in string and array operations. It holds the address from where to read data.

EDI: Destination Index register. Pointer to data (or destination) in the segment pointed to by the ES register. Used as an offset address in string and array operations. It holds the implied write address of all string operations.

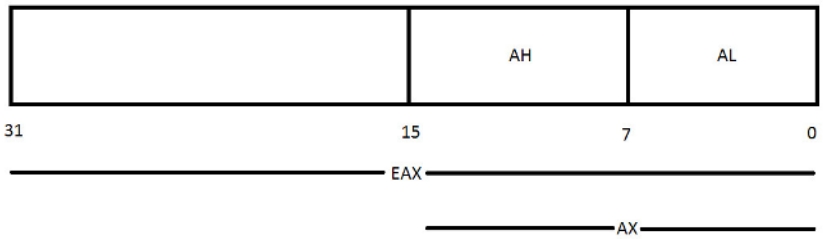
EBP: Base Pointer. Pointer to data on the stack (in the SS segment). It points to the bottom of the current stack frame. It is used to reference local variables.

ESP: Stack Pointer (in the SS segment). It points to the top of the current stack frame. It is used to reference local variables.

Keep in mind each of the above registers are 32-bit in length or 4 bytes in length. Each of the lower 2 bytes of the EAX, EBX, ECX, and EDX registers can be referenced by AX and then subdivided by the names AH, BH, CH and DH for high bytes and AL, BL, CL and DL for the low bytes which are 1 byte each.

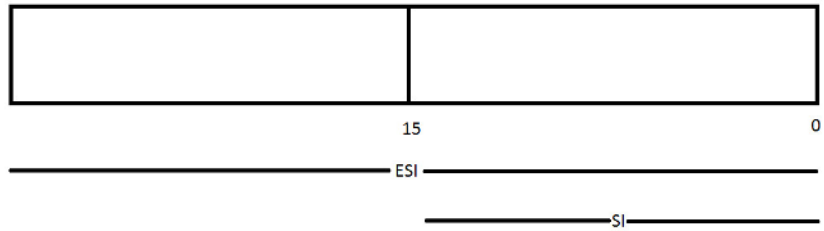
In addition, the ESI, EDI, EBP and ESP can be referenced by their 16-bit equivalent which is SI, DI, BP, SP.

This can be a bit confusing to someone who has not studied computer engineering however let me illustrate in the table below:



EAX would have AX as its 16-bit segment and then you can further subdivide AX into AL for the low 8 bits and AH for the high 8 bits. The same holds true for EBX, ECX and EDX as well. EBX would have BX as its 16-bit segment and then you can further subdivide BX into BL for the low 8 bits and BH for the high 8 bits. ECX would have CX as its 16-bit segment and then you can further subdivide CX into CL for the low 8 bits and CH for the high 8 bits. EDX would have DX as its 16-bit segment and then you can further subdivide DX into DL for the low 8 bits and DH for the high 8 bits.

ESI, EDI, EBP and ESP can be broken down into its 16-bit segments as follows:



ESI would have SI as its 16-bit segment, EDI would have DI as its 16-bit segment, EBP would have BP as its 16-bit segment and ESP would have SP as its 16-bit segment.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the Segment Registers.

Part 11: Segment Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The segment registers are used specifically for referencing memory locations. There are three different methods of accessing system memory of which we will focus on the flat memory model which is relevant for our purposes.

There are six segment registers which are as follows:

CS: Code segment register stores the base location of the code section (.text section) which is used for data access.

DS: Data segment register stores the default location for variables (.data section) which is used for data access.

ES: Extra segment register which is used during string operations.

SS: Stack segment register stores the base location of the stack segment and is used when implicitly using the stack pointer or when explicitly using the base pointer.

FS: Extra segment register.

GS: Extra segment register.

Each segment register is 16-bits and contains the pointer to the start of the memory-specific segment. The CS register contains the pointer to the code segment in memory. The code segment is where the instruction codes are stored in memory. The processor retrieves instruction codes from memory based on the CS register value and an offset value contained in the instruction pointer (EIP) register. Keep in mind no program can explicitly load or change the CS register. The processor assigns its values as the program is assigned a memory space.

The DS, ES, FS and GS segment registers are all used to point to data segments. Each of the four separate data segments help the program separate data elements to ensure that they do no overlap. The program loads the data segment registers with the appropriate pointer value for the segments and then reference individual memory locations using an offset value.

The stack segment register (SS) is used to point to the stack segment. The stack contains data values passed to functions and procedures within the program.

Segment registers are considered part of the operating system and can neither read nor be changed directly in almost all cases. When working in the protected mode flat model (x86 architecture which is 32-bit), your program runs and receives a 4GB address space to which any 32-bit register can potentially address any of the four billion memory locations except for those protected areas defined by the operating system. Physical memory may be larger than 4GB however a 32-bit register can only express 4,294,967,296 different locations. If you have more than 4GB of memory in your computer, the OS must arrange a

4GB region within memory and your programs are limited to that new region. This task is completed by the segment registers and the OS keeps close control of this.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the Instruction Pointer Register.

Part 12: Instruction Pointer Register

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The instruction pointer register called the EIP register is simply the most important register you will deal with in any reverse engineering. The EIP keeps track of the next instruction code to execute. EIP points to the next instruction to execute. If you were to alter that pointer to jump to another area in the code you have complete control over that program.

Lets jump ahead and dive into some code. Here is an example of a simple hello world application in C that we will go into more detail much later in our tutorial series. For our purposes today, we will see the raw POWER of assembly language and particularly that of the EIP register and what we can do to completely hack program control.

```c
#include <stdio.h>
#include <stdlib.h>

void unreachableFunction(void) {
    printf("I'm hacked!  I am a hidden function!\n");
    exit(0);
}

int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

Don't worry if you do not understand what it does or its functionality. What to take note of here is the fact we have a function called unreachableFunction that is never called by the main function. As you will see if we can control the EIP register we can hack this program to execute that code!

```
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o eipExample eipExample.c
noroot@noroot-VirtualBox:~/Desktop$ nano eipExample.c
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o eipExample eipExample.c
noroot@noroot-VirtualBox:~/Desktop$ ./eipExample
Hello World!
```

We have simply compiled the code to work with the IA32 instruction set and ran it. As you can see there is no call to the **unreachableFunction** of any kind as it is unreachable under normal conditions as you can see the 'Hello World!` printed when executed.

```
(gdb) set disassembly-flavor intel
(gdb) b main
Breakpoint 1 at 0x804846c: file eipExample.c, line 10.
(gdb) r
Starting program: /home/noroot/Desktop/eipExample

Breakpoint 1, main () at eipExample.c:10
10          printf("Hello World!\n");
(gdb) disas
Dump of assembler code for function main:
   0x0804845b <+0>:     lea    ecx,[esp+0x4]
   0x0804845f <+4>:     and    esp,0xfffffff0
   0x08048462 <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048465 <+10>:    push   ebp
   0x08048466 <+11>:    mov    ebp,esp
   0x08048468 <+13>:    push   ecx
   0x08048469 <+14>:    sub    esp,0x4
=> 0x0804846c <+17>:    sub    esp,0xc
   0x0804846f <+20>:    push   0x8048535
   0x08048474 <+25>:    call   0x8048300 <puts@plt>
   0x08048479 <+30>:    add    esp,0x10
   0x0804847c <+33>:    mov    eax,0x0
   0x08048481 <+38>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x08048484 <+41>:    leave
   0x08048485 <+42>:    lea    esp,[ecx-0x4]
   0x08048488 <+45>:    ret
End of assembler dump.
```

We have disassembled the program using the GDB Debugger. We have set a breakpoint on the main function and ran the program. The => shows where EIP is pointing to when we step to the next instruction. If we follow normal program flow, ‘Hello World! will print to the console and exit.

```
(gdb) c
Continuing.
Hello World!
[Inferior 1 (process 3034) exited normally]
```

If we run the program again and do an examination of where EIP is pointing to we will see:

```
(gdb) r
Starting program: /home/noroot/Desktop/eipExample

Breakpoint 1, main () at eipExample.c:10
10          printf("Hello World!\n");
(gdb) x/1xb $eip
0x804846c <main+17>:    0x83
(gdb) x/1xw $eip
0x804846c <main+17>:    0x680cec83
```

We can see EIP is pointing to main+17 or the address of 0x680cec83.

Lets examine the **unreachableFunction** and see where it starts in memory and write down that address.

```
(gdb) disas unreachableFunction
Dump of assembler code for function unreachableFunction:
   0x0804843b <+0>:     push   ebp
   0x0804843c <+1>:     mov    ebp,esp
   0x0804843e <+3>:     sub    esp,0x8
   0x08048441 <+6>:     sub    esp,0xc
   0x08048444 <+9>:     push   0x8048510
   0x08048449 <+14>:    call   0x8048300 <puts@plt>
   0x0804844e <+19>:    add    esp,0x10
   0x08048451 <+22>:    sub    esp,0xc
   0x08048454 <+25>:    push   0x0
   0x08048456 <+27>:    call   0x8048310 <exit@plt>
End of assembler dump.
```

The next step is to set EIP to address 0x0804843b so that we hijack program flow to run the unreachableFunction.

```
(gdb) set $eip = 0x0804843b
(gdb) x/1xw $eip
0x804843b <unreachableFunction>:        0x83e58955
```

Now that we have hacked control of EIP, lets continue and watch how we have hijacked the operation of a running program to our advantage!

```
(gdb) c
Continuing.
I'm hacked!  I am a hidden function!
[Inferior 1 (process 3048) exited normally]
```

Tada! We have hacked the program!

So the question in your mind is why did you show me this when I have no idea of what any of this is? It is important to understand that when we are doing a lengthy tutorial such as this we should sometimes look forward to see why we are taking so many steps to learn the basics before we dive in. It is important however to show you that if you stay with the tutorial your hard work will pay off as we will learn how to hijack any running program to make it do whatever we want in addition to proactively breaking down a malicious program so that we can not only disable it but trace it back to a potential IP of where the hack originated.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the Control Registers.

Part 13: Control Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Their are five control registers which are used to determine the operating mode of the CPU and the characteristics of the current executing task. Each control register is as follows:

CR0: System flag that control the operating mode and various states of the processor.

CR1: (Not Currently Implemented)

CR2: Memory page fault information.

CR3: Memory page directory information.

CR4: Flags that enable processor feathers and indicate feature capabilities of the processor.

The values in each of the control registers can't be directly accessed however the data in the control register can be moved to one of the general-purpose registers and once the data is in a GP register, a program can examine the bit flags in the register to determine the operating status of the processor in conjunction with the current running task.

If a change is required to a control register flag value, the change can be made to the data in the GP register and the register moved to the CR. Low-level System Programmers usually modify the values in control registers. Normal application programs do not usually modify control register entries however they might query flag values to determine the capabilities of the host processor chip on which the program is currently running.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the topic of Flags.

Part 14: Flags

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The topic of flags are one of the most extremely complex and complicated concepts of assembly language and program flow control when reverse engineering. This information below will become much clearer as we enter into the final phase of our training when we reverse engineer C applications into assembly language.

What is important here is to take away the fact that flags help control, check and verify program execution and are a mechanism to determine whether each operation that is performed by the processor is successful or not.

Flags are critical to assembly language applications as they are a check to verify each programs functions successful execution.

We are dealing with 32-bit assembly to which a single 32-bit register which contains a group of status, control and system flags exist. This register is called the EFLAGS register as it contains 32 bits of information that are mapped to represent specific flags of information.

There are three kinds of flags which are status flags, control flags and system flags.

Status flags are as follows:

CF: Carry Flag

PF: Parity Flag

AF: Adjust Flag

ZF: Zero Flag

SF: Sign Flag

OF: Overflow Flag

The carry flag is set when a math operation on an unsigned integer value generates a carry or borrow for the most significant bit. This is an overflow condition for the register involved in the math operation. When this occurs, the remaining data in the register is not the correct answer to the math operation.

The parity flag is used to indicate corrupt data as a result of a math operation in a register. When checked, the parity flag is set if the total number of 1 bits in the result is even and is cleared if the total number of 1 bits in the result is odd. When the parity flag is checked, an application can determine whether the register has been corrupted since the operation.

The adjust flag is used in Binary Coded Decimal math operations and is set if a carry or borrow operation occurs from bit 3 of the register used for the calculation.

The zero flag is set if the result of an operation is zero.

The sign flag is set to the most significant bit of the result which is the sign bit and indicates whether the result is positive or negative.

The overflow flag is used in signed integer arithmetic when a positive value is too big or a negative value is too small to be represented in the register.

Control flags are utilized to control specific behavior in the processor. The DF flag which is the direction flag is used to control the way strings are handled by the processor. When set, string instructions automatically decrement memory addresses to get the next byte in the string. When cleared, string instructions automatically increment memory addresses to get the next byte in the string.

System flags are used to control OS level operations which should NEVER be modified by any respective program or application.

TF: Trap Flag

IF: Interrupt Enable Flag

IOPL: I/O Privilege Level Flag

NT: Nested Task Flag

RF: Resume Flag

VM: Virtual-8086 Mode Flag

AC: Alignment Check Flag

VIF: Virtual Interrupt Flag

VIP: Virtual Interrupt Pending Flag

ID: Identification Flag

The trap flag is set to enable single-step mode and when in this mode the processor performs only one instruction code at a time, waiting for a signal to perform the next instruction. This is essential when debugging.

The interrupt enable flag controls how the processor responds to signals received from external sources.

The I/O privilege field indicates the input-output privilege level of the currently running task and defines access levels for the input-output address space which must be less than or equal to the access level required to access the respective address space. In the case where it is not less than or equal to the access level required, any request to access the address space will be denied.

The nested task flag controls whether the currently running task is linked to the previously executed task and is used for chaining interrupted and called tasks.

The resume flag controls how the processor responds to exceptions when in debugging mode.

The VM flag indicates that the processor is operating in virtual-8086 mode instead of protected or real mode.

The alignment check flag is used in conjunction with the AM bit in the CR0 control register to enable alignment checking of memory references.

The virtual interrupt flag replicates the IF flag when the processor is operating in virtual mode.

The virtual interrupt pending flag is used when the processor is operating in virtual mode to indicate that n interrupt is pending.

The ID flag indicates whether the processor supports the CPUID instruction.

In our next tutorial we will discuss the stack.

Part 15: Stack

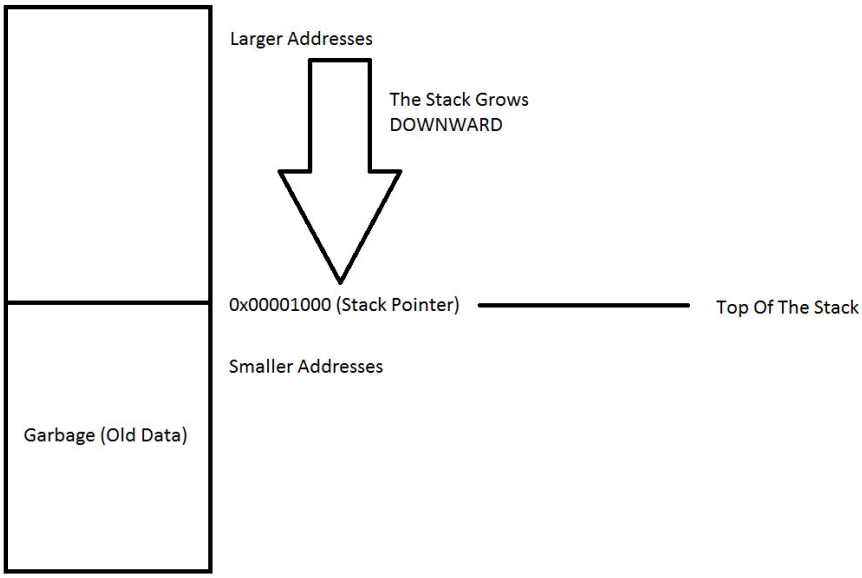
For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Functions are the most fundamental feature in software development. A function allows you to organize code in a logical way to execute a specified task. It is not critical that you understand how functions work at this stage it is only important that you understand that when we start learning to develop, we want to minimize duplication by using functions that can be called multiple times rather than duplicate code taking up excessive memory.

When a program starts to execute a certain contiguous section of memory is set aside for the program called the stack.

The stack pointer is a register that contains the top of the stack. The stack pointer contains the smallest address, lets say for example 0x00001000, such that any address smaller than 0x00001000 is considered garbage and any address greater than 0x00001000 is considered valid.

The above address is random and is not an absolute where you will find the stack pointer from program to program as it will vary. Lets look at what the stack looks like from an abstract perspective:



The above diagram is what I want you to keep clear in your mind as that is what is actually happening in memory. The next series of diagrams will show the opposite of what is shown above.

You will see the stack growing upward in the below diagrams however in reality it is growing downward from higher memory to lower memory.

In the addMe example below, the stack pointer (ESP), when examined in memory on a breakpoint on the main function, lists 0xffffd050. When the program calls the addMe function from main, ESP is now 0xffffd030 which is LOWER in memory.

Therefore the stack grows DOWNWARD despite the diagram showing it pointing upward. Just keep in mind when the arrows below are pointing upward they are actually pointing to lower memory addresses.

The stack bottom is the largest valid address of the stack and is located in the larger address area or top of the memory model. This can be confusing as the stack bottom is higher in memory. The stack grows downward in memory and it is critical that you understand that now as we go forward.

The stack limit is the smallest valid address of the stack. If the stack pointer gets smaller than this, there is a stack overflow which can corrupt a program to allow an attacker to take control of a system. Malware attempts to take advantage of stack overflows. As of recent, there are protections build into modern OS that attempt to prevent this from happening.

There are two operations on the stack which are push and pop. You can push one or more registers by setting the stack pointer to a smaller value. This is usually done by subtracting four times the number of registers to be pushed onto the stack and copying the registers to the stack.

You can pop one or more registers by copying the data from the stack to the registers, then to add a value to the stack pointer. This is usually done by adding four times the number of registers to be popped on the stack.

Let us look at how the stack is used to implement functions. For each function call there is a section of the stack reserved for the function. This is called the stack frame.

Let's look at the C program we created in tutorial 12 and examine what the main function looks like:

```c
#include <stdio.h>
#include <stdlib.h>

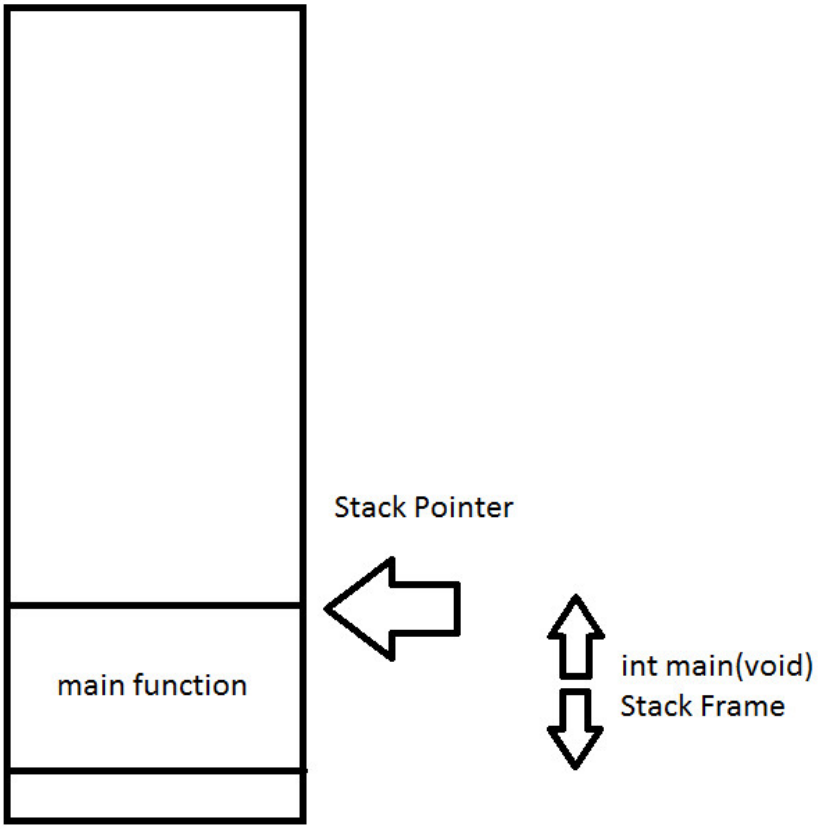
void unreachableFunction(void) {
    printf("I'm hacked!  I am a hidden function!\n");
    exit(0);
}

int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

We see two functions here. The first one is the unreachableFunction to which will never execute under normal circumstances and we also see the main function that will always be the first function to be called onto the stack.

When we run this program, the stack will look like this:



We can see the stack frame for int main(void) above. It is also referred to as the activation record. A stack frame exists whenever a function has started but yet to complete. For example, inside of the body of the int main(void) there is a call to int addMe(int a, int b) which takes two arguments a and b. There needs to be assembly language code in int main(void) to push the arguments for int addMe(int a, int b) onto the stack. Lets examine some code.

```
#include <stdio.h>

int addMe(int a, int b);

int main(void){
    int result = addMe(2, 3);

    printf("The result of the addMe function is %d!\n", result);

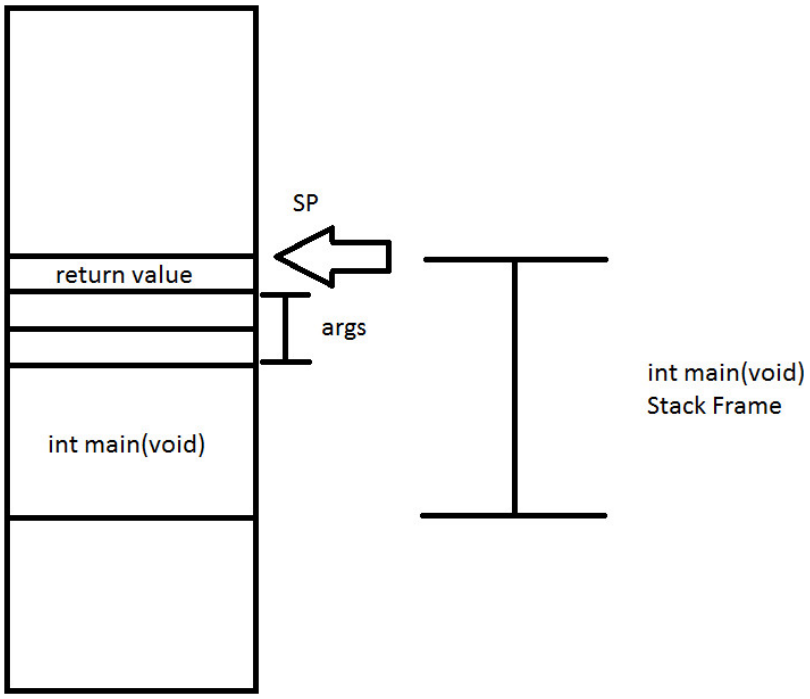
    return 0;
}

int addMe(int a, int b){
    return a + b;
}
```

When we compile and run this program we will see the value of 5 to be print out like this:

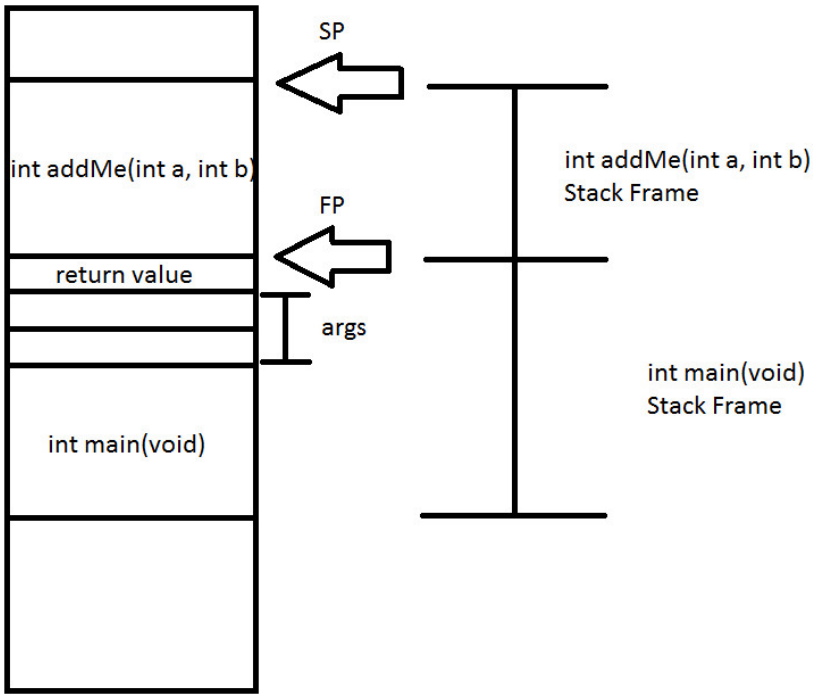
```
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o addMe addMe.c
noroot@noroot-VirtualBox:~/Desktop$ ./addMe
The result of the addMe function is 5!
```

Very simply, int main(void) calls int addMe(int a, int b) first and will get put on the stack like this:



You can see that by placing the arguments on the stack, the stack frame for **int main(void)** has increased in size. We also reserved space for the return value which is computed by **int addMe(int a, int b)** and when the function returns, the return value in **int main(void)** gets restored and execution continues in **int main(void)** until it finishes.

Once we get the instructions for **int addMe(int a, int b)**, the function may need local variables so the function needs to push some space on the stack which would look like:



int addMe(int a, int b) can access the arguments passed to it from **int main(void)** because the code in **int main(void)** places the arguments just as **int addMe(int a, int b)** expects it.

FP is the frame pointer and points to the location where the stack pointer was just before **int addMe(int a, int b)** moved the stack pointer or SP for int **addMe(int a, int b)**'s own local variables.

The use of a frame pointer is essential when a function is likely to move the stack pointer several times throughout the course of running the function. The idea is to keep the frame pointer fixed for the duration of **int addMe(int a, int b)**'s stack frame. In the meantime, the stack pointer can change values.

We can use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it does not move, the computations for those locations should be some fixed offset from the frame pointer.

Once it is time to exit **int addMe(int a, int b)**, the stack pointer is set to where the frame pointer is which pops off the **int addMe(int a, int b)** stack frame.

In sum, the stack is a special region of memory that stores temporary variables created by each function including main. The stack is a LIFO which is last in, first out data structure which is managed and optimized by the CPU closely. Every time a function declares a new variable it is pushed onto the stack. Every time a function exists, all of the variables pushed onto the stack by that function are freed or deleted. Once a stack variable is freed, that region of memory becomes available for other stack variables.

The advantage of the stack to store variables is that memory is managed for you. You do not have to allocate memory manually or free it manually. The CPU manages and organizes stack memory very efficiently and is very fast.

It is critical that you understand that when a function exits, all of its variables are popped off the stack and lost forever. The stack variables are local. The stack grows and shrinks as functions push and pop local variables.

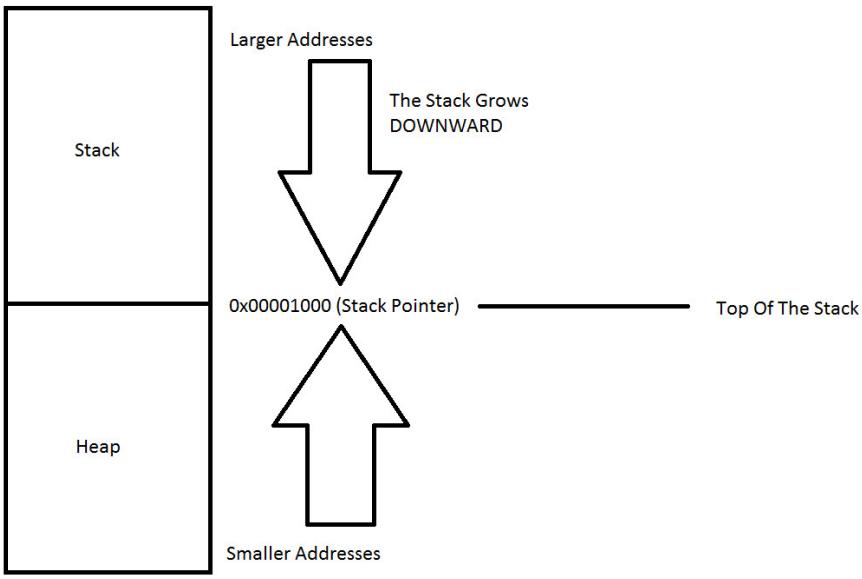
I can see your head spinning around and around. Keep in mind, these topics are complicated and will continue to develop in future tutorials. We have been dealing with a lot of confusing topics such as registers, memory and now the stack and it can be overwhelming. If you ever have questions, please comment below and I will help you to better understand this framework.

In our next tutorial we will discuss the heap.

Part 16: Heap

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Our next step in the Basic Malware Reverse Engineering section focuses on the heap. Keep in mind, the stack grows downward and the heap grows upward. It is very, very important that you understand this concept as we progress forward in our future tutorials.



The heap is the region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is free-floating region of memory and is larger than the stack allocation of memory.

To allocate memory on the heap, you must use **malloc()** or **calloc()**, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for freeing it by using **free()** to de-allocate that memory once you don't need it any more.

If you don’t do this step, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside and won't be available to other processes that need it.

Unlike the stack, the heap does not have size restrictions on variable size. The only thing that would limit the heap is the physical limitations of your computer. Heap memory is slightly slower to be read from and written to, because you have to to use pointers to access memory on the heap. When we dive into our C tutorial series we will demonstrate this.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

If you need to allocate a large block of memory for something like a struct or a large array and you need to keep that variable around for a good duration of the program to which must be accessed globally, then you should choose the heap for this purpose. If you need variables like arrays and structs that can change size dynamically such as arrays that can grow or shrink as needed, then you will likely need to allocate them on the heap, and use dynamic memory allocation functions like **malloc()**, **calloc()**, **realloc()** and **free()** to manage that memory manually.

The next step is to dive into programming C in the Linux environment where we step-by-step disassemble each C program so in effect you will be learning both C programming and Assembly so that you can progress your skills in Malware Analysis and Reverse Engineering.

I look forward to seeing you all next week when we take a comprehensive step-by-step tutorial on how to install Linux on your current computer using the FREE Virtual Box software tool.

Part 17 – How To Install Linux

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

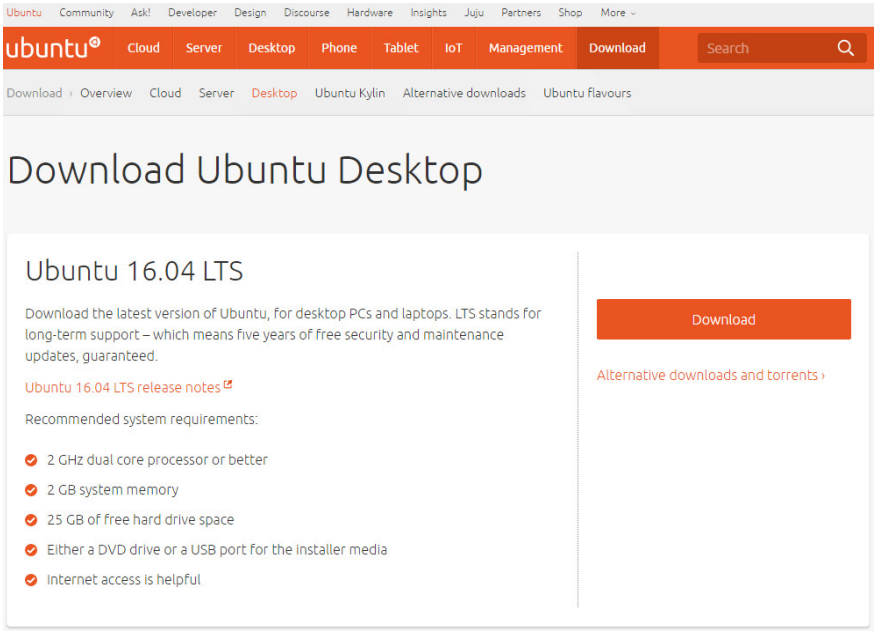
If you do not have Linux installed on a computer within your household, I would suggest installing Virtual Box which is an open-source free virtual environment which you can install on your existing computer to have a version of Linux you can program with. Below is a link to download and install Virtual Box as there are versions for both Windows and Mac.

https://www.virtualbox.org/wiki/Downloads



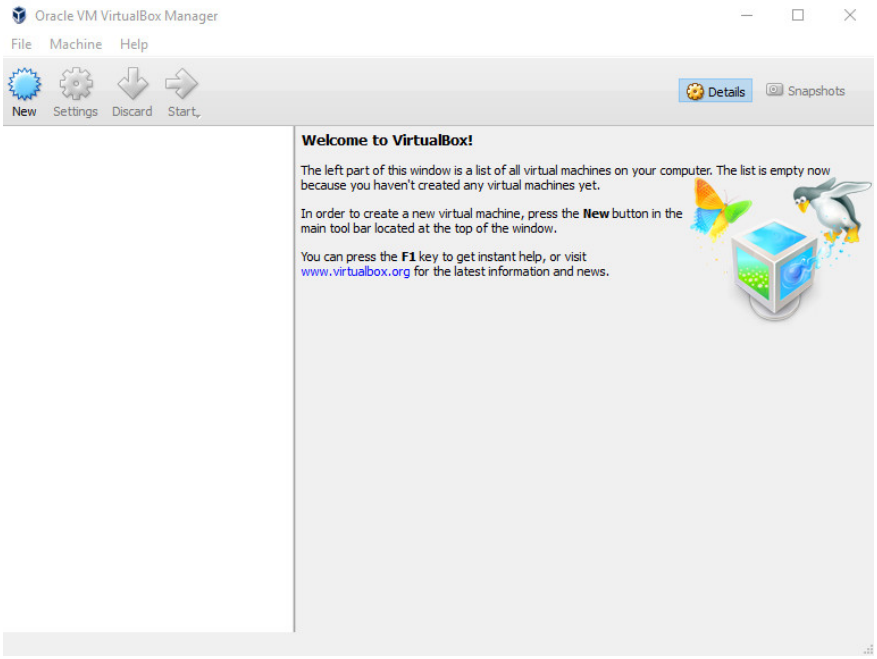
In addition, you will need a copy of Linux to which I will be working with Ubuntu. Below is a link to download the .iso file to which you will install once you have Virtual Box installed.

http://www.ubuntu.com/download/desktop

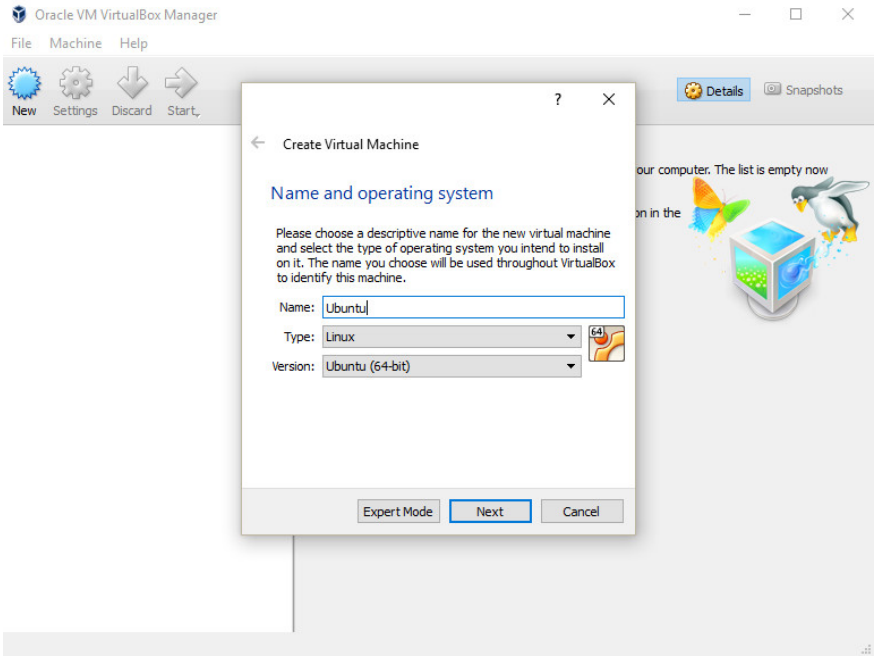


After you download the above **.iso**, go to your **Download** directory and first execute and run the **VirtualBox-5.0.24-108355-Win.exe** or whatever version of VirtualBox that is currently available. If you are running a Mac, you will download the **.dmg** file. Simply double-click on the file to execute and run it.

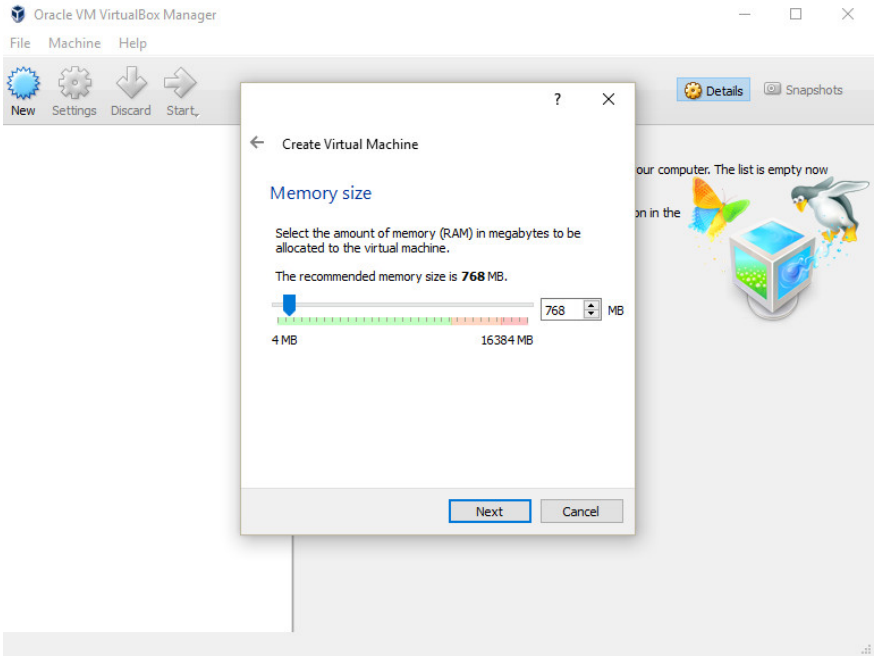
After you install **VirtualBox-5.0.24-108355-Win.exe** or the Mac **.dmg** file and you will see this screen:



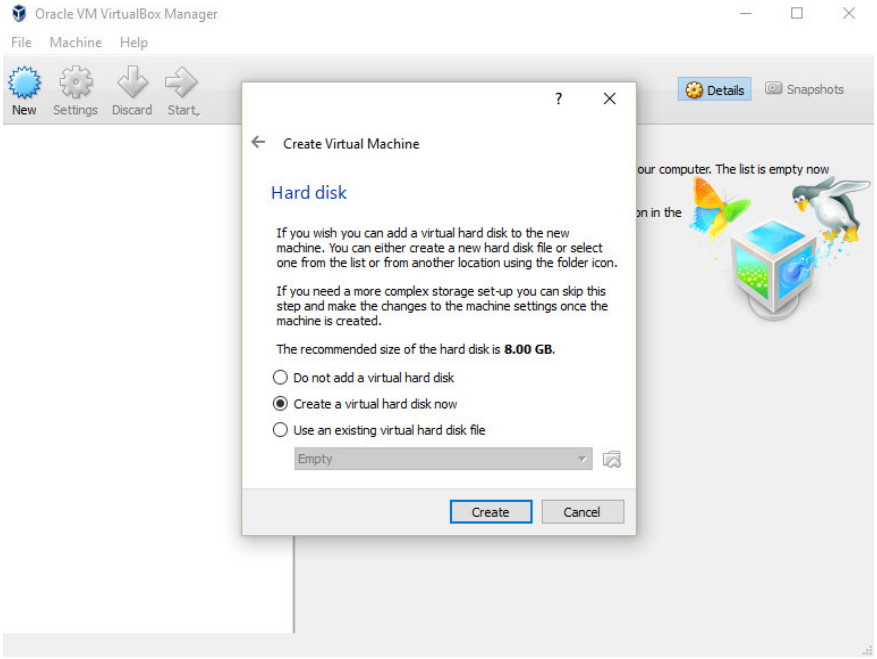
Click on the New button above which is located in the top-left corner of the screen as it is a big blue cog-looking circle.



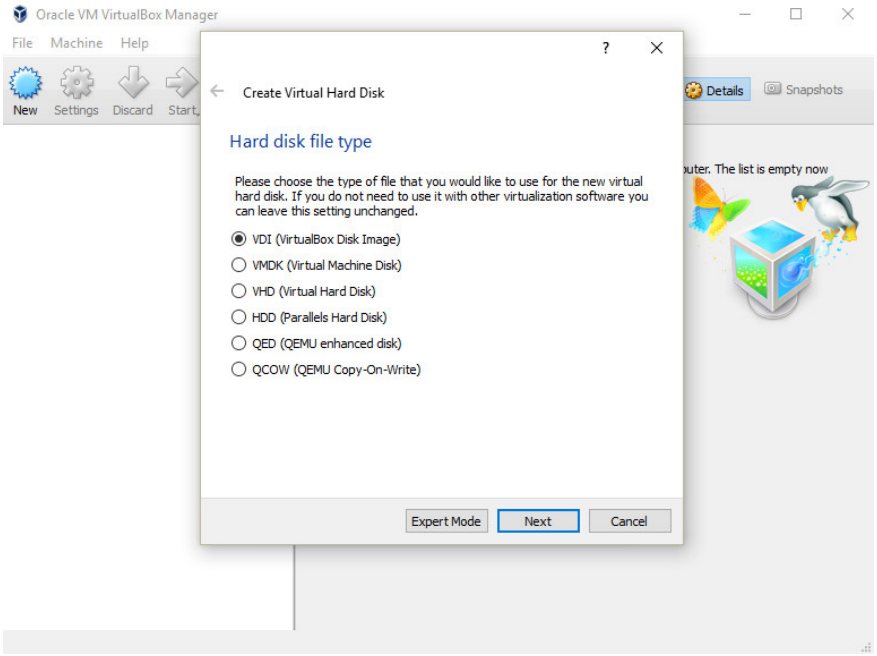
In the name field above, type Ubuntu and click the next button.



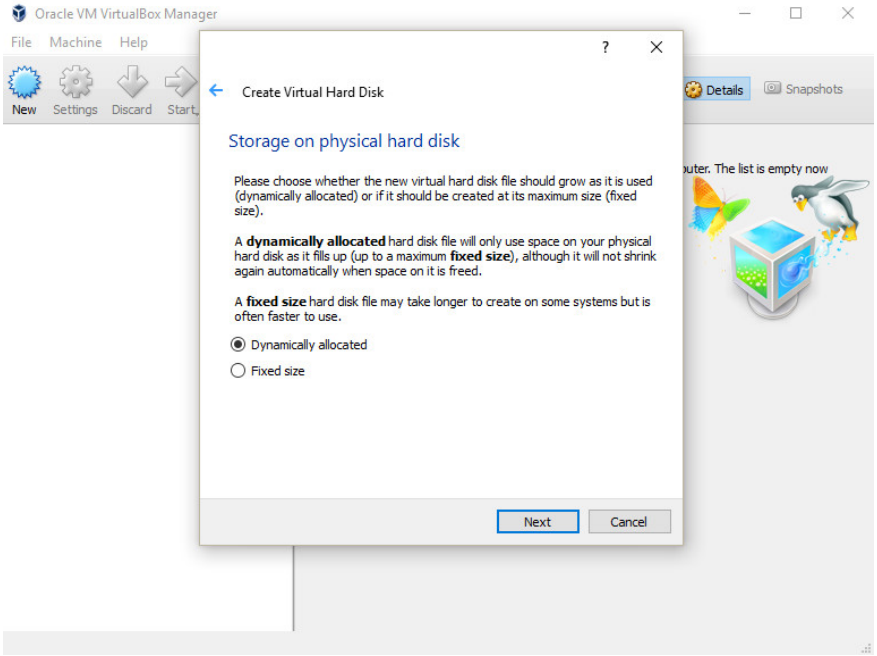
It is important to click on the blue slider bar above and select an amount of ram that points to an area in green so that it does not overwhelm your computer resources. After moving the blue slider, click next.



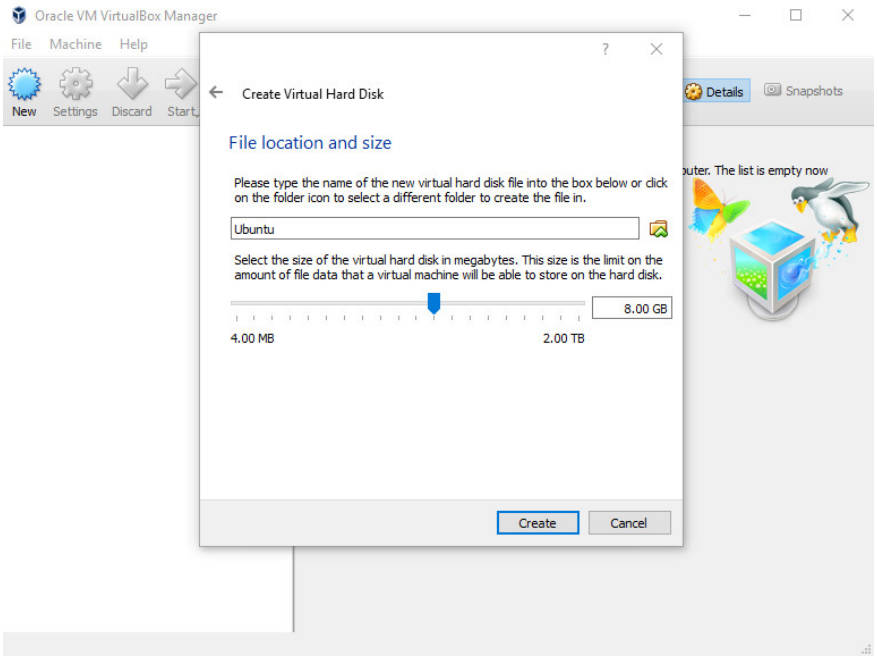
Then click create.



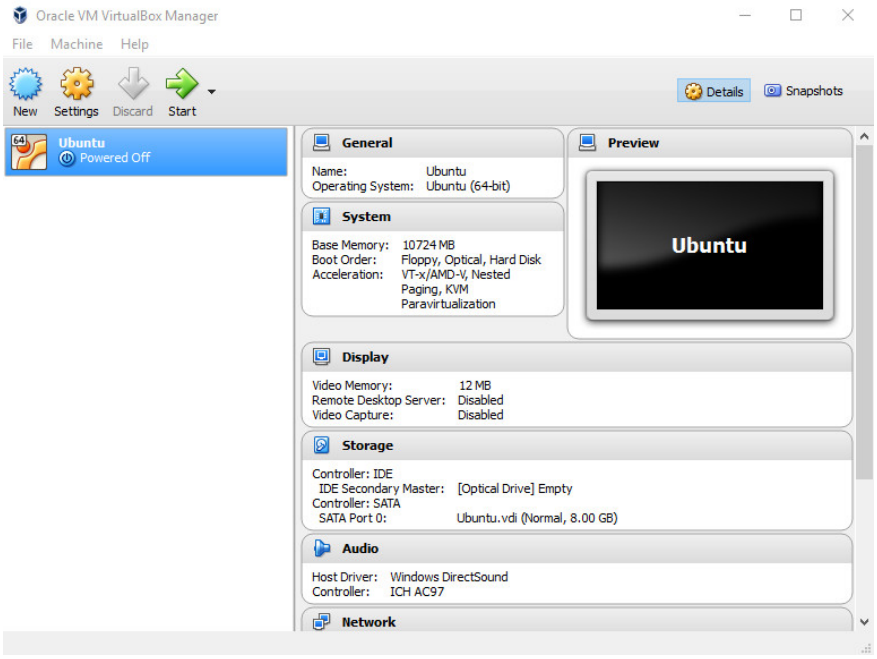
Then click next.



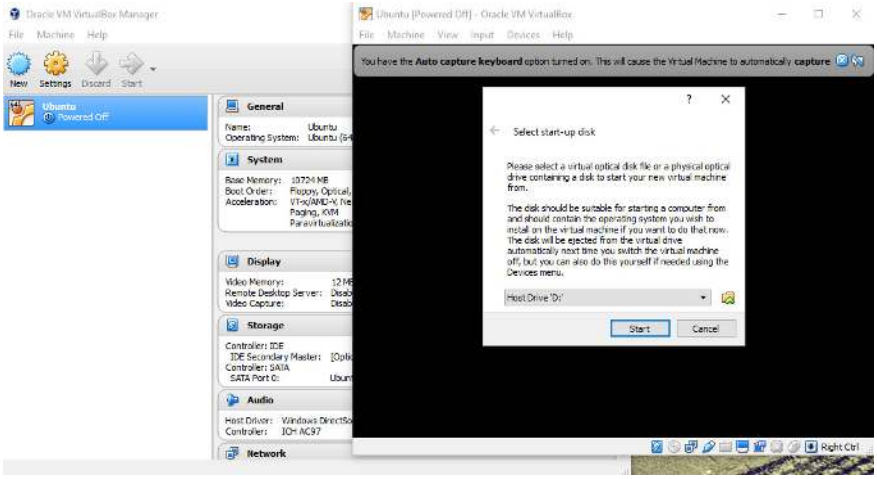
Then click next.



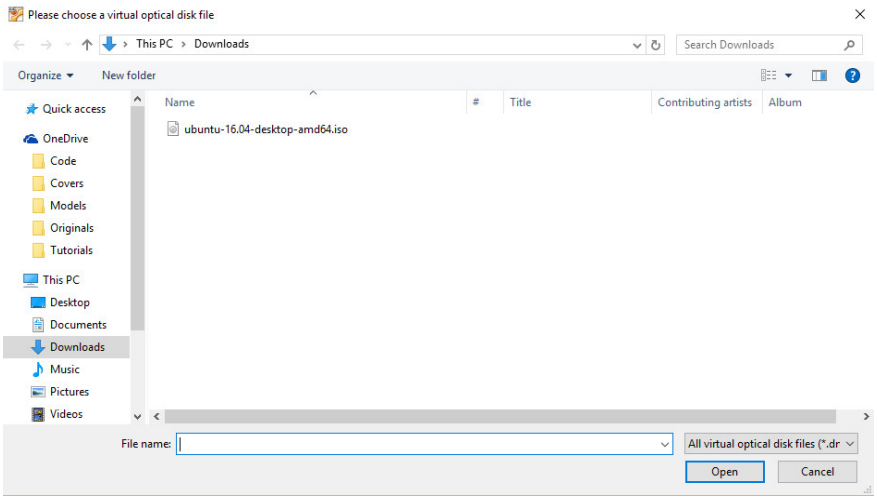
Please move the dial up to 16.00 GB rather than 8.00 GB shown above then click create.



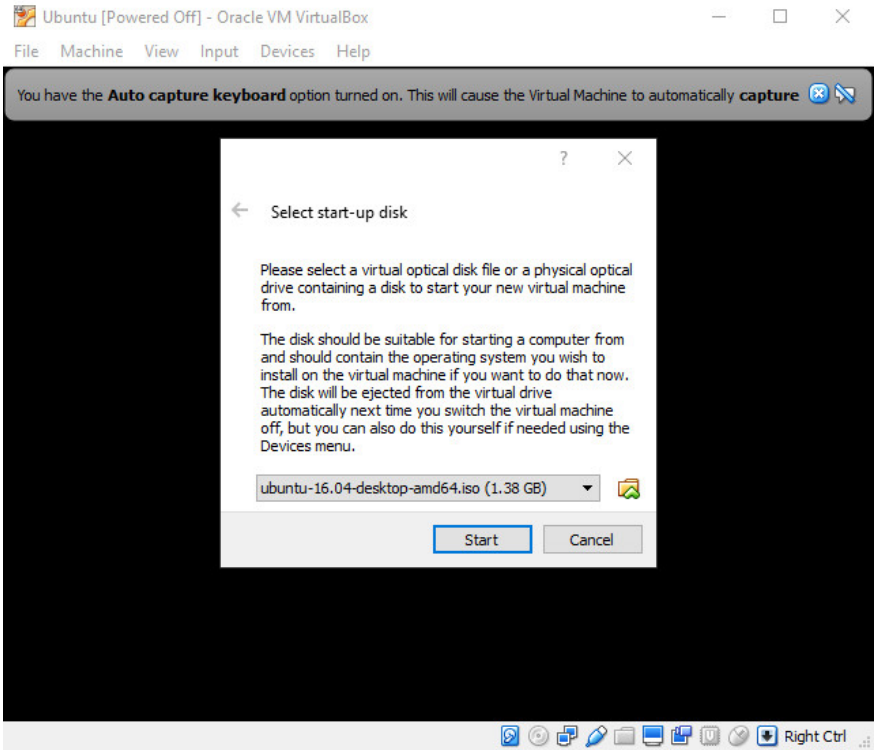
The next step is to click on the green start button.



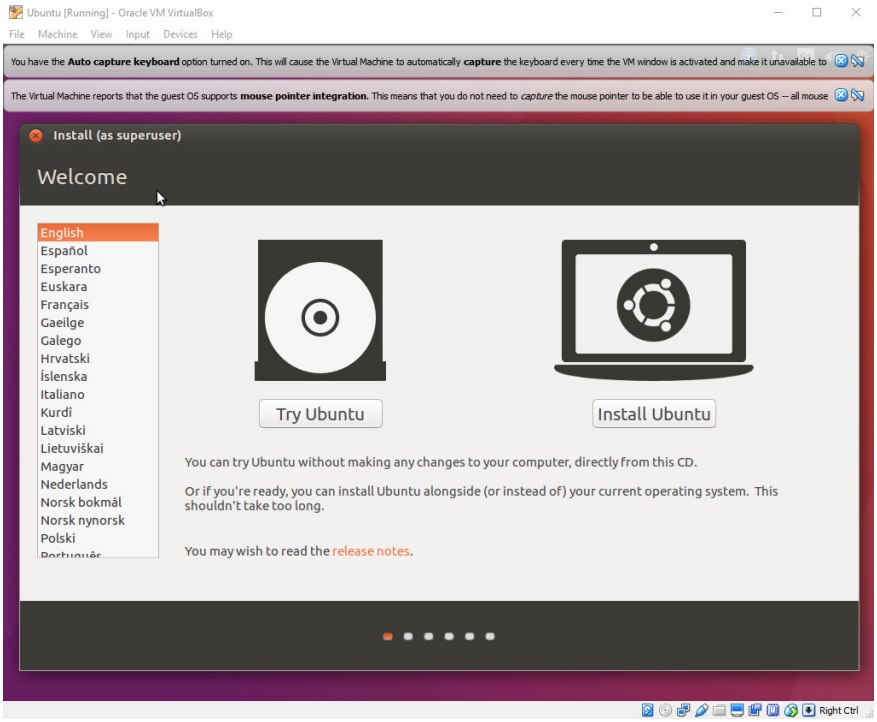
The next step is to click on the yellow folder just above the cancel button.



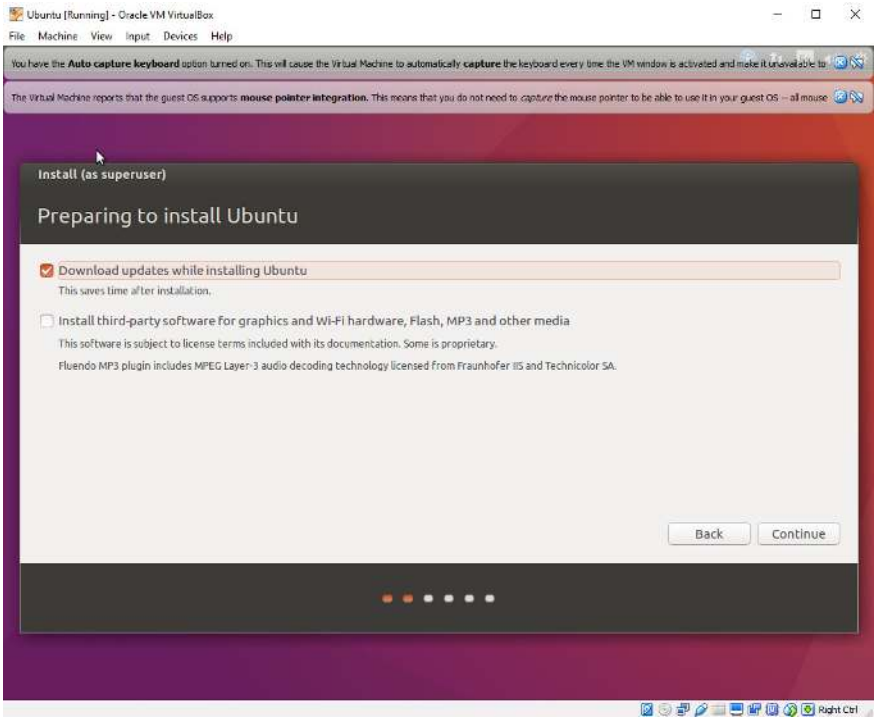
The next step is to click on the .iso file that should be in your Download directory and click open.



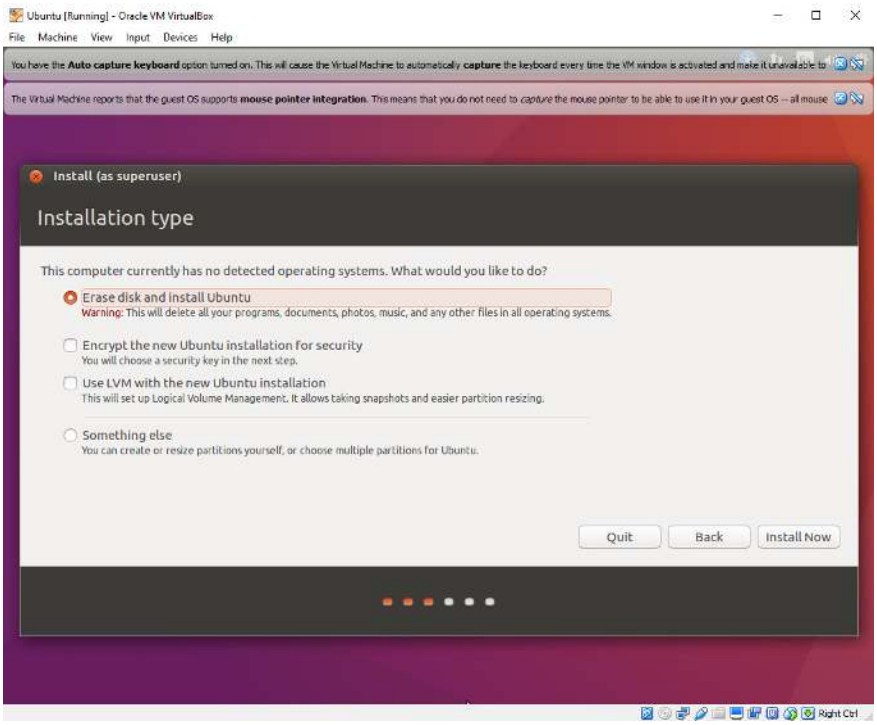
The next step is to click start.



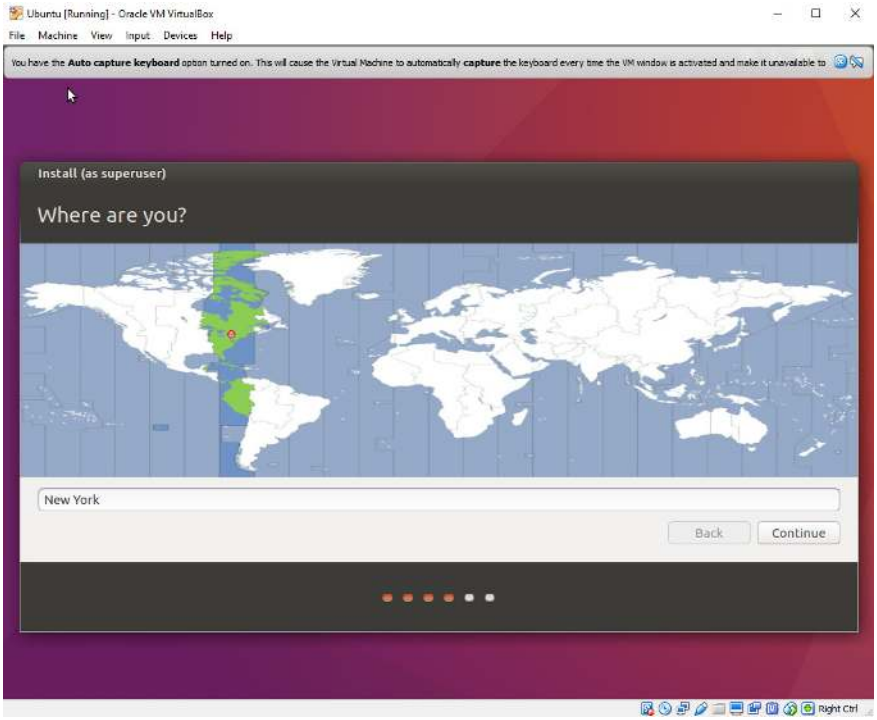
The next step is to let the install begin and click Install Ubuntu.



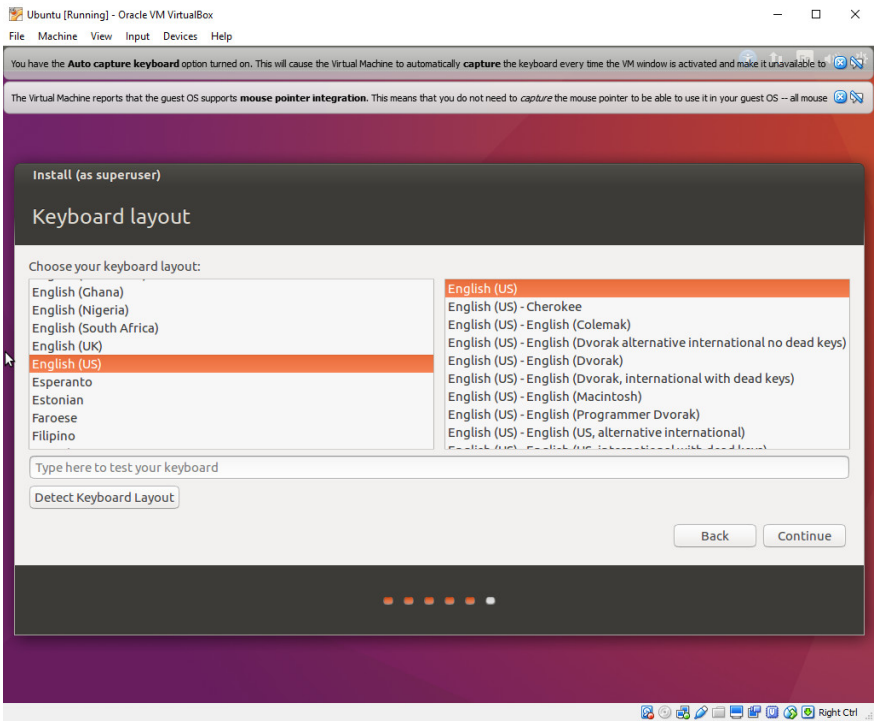
The next step is to check each of the boxes to Download updates while installing Ubuntu and click continue.



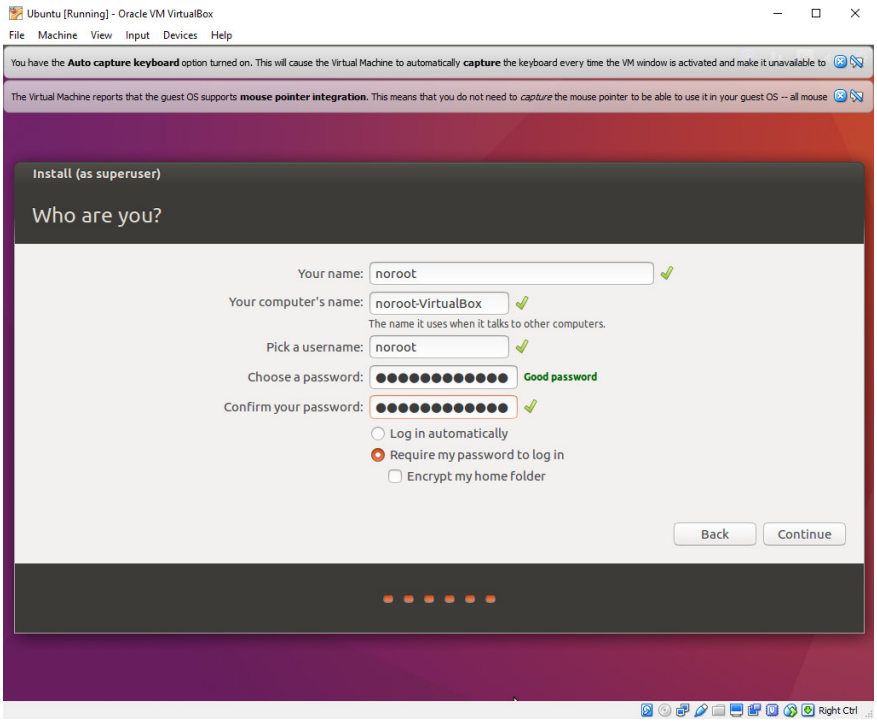
The next step is to select Erase disk and install Ubuntu and click install now.



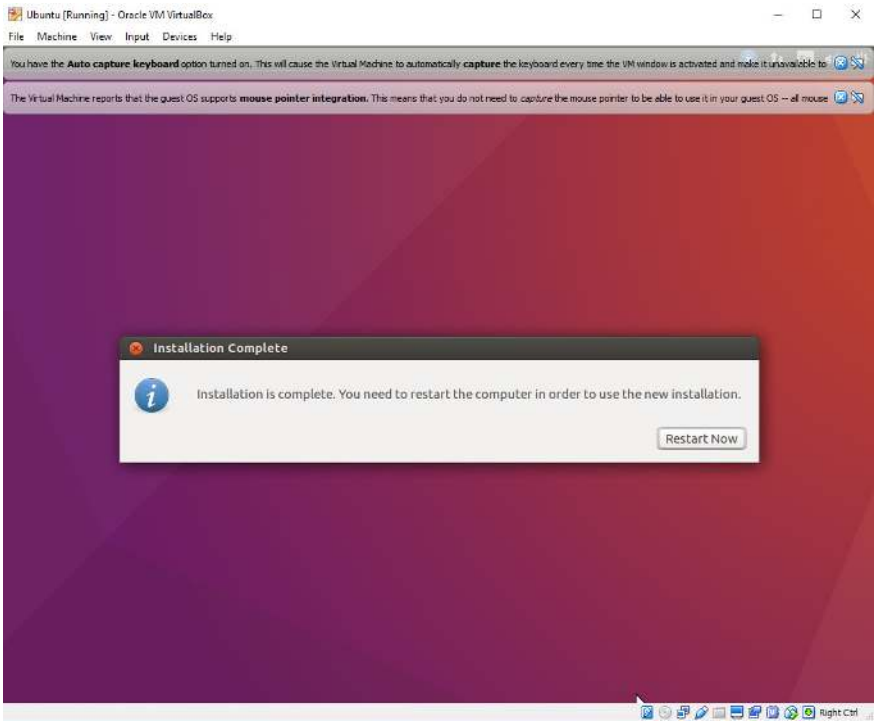
The next step is to click continue and progress forward to the screen where you will select your timezone to which you will select continue.



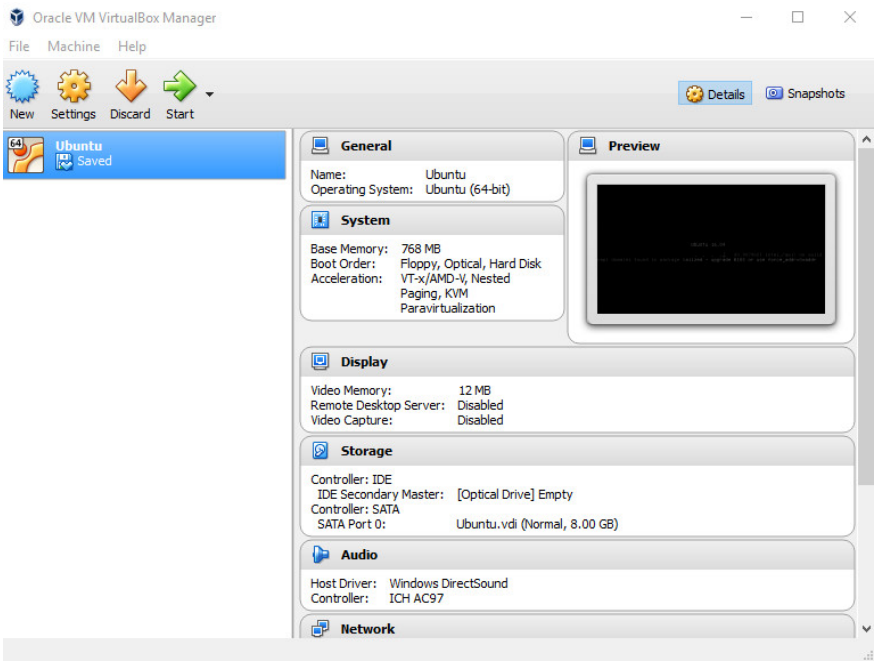
The next step is to select your keyboard layout and click continue.



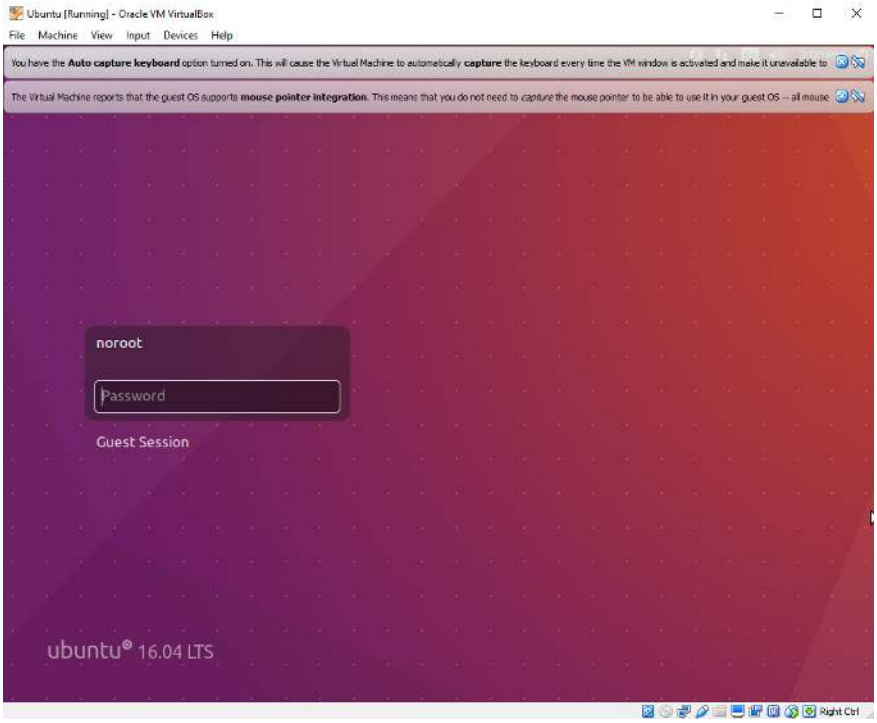
The next step is to create a name for your account. I chose noroot and did the same for the username. In addition, create a password and re-type it for verification and click continue.



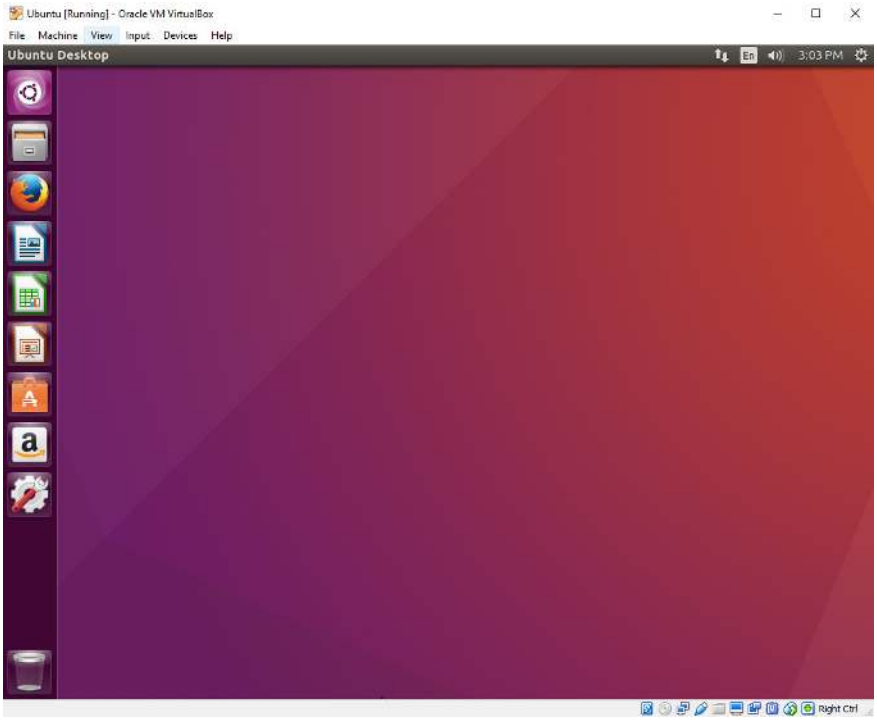
At this point it will take some time to install the operating system. When the process is finished, click restart now. If the window locks up, click Power Off The Machine and click close or next.



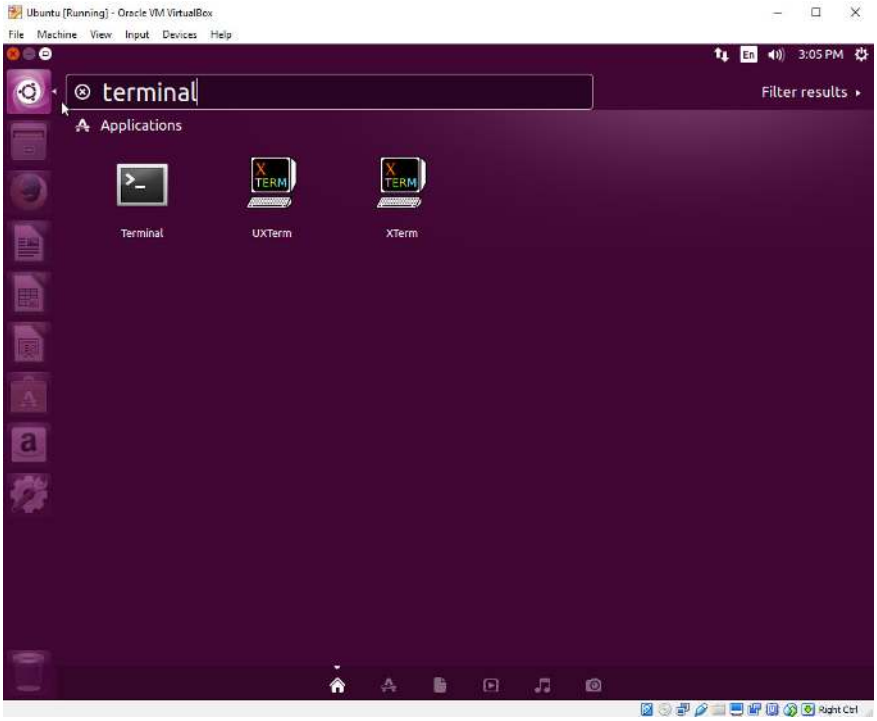
At this point, click on the green start button.



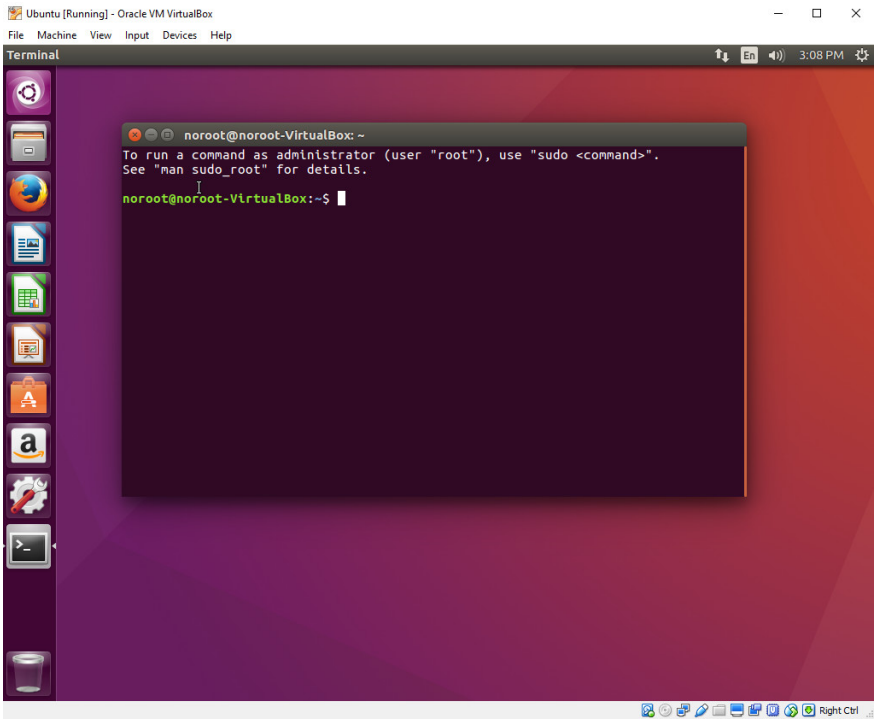
Enter in your password that you created earlier and click enter on your keyboard. You can click on the blue x buttons in the top right corner as they are just some information you can close out.



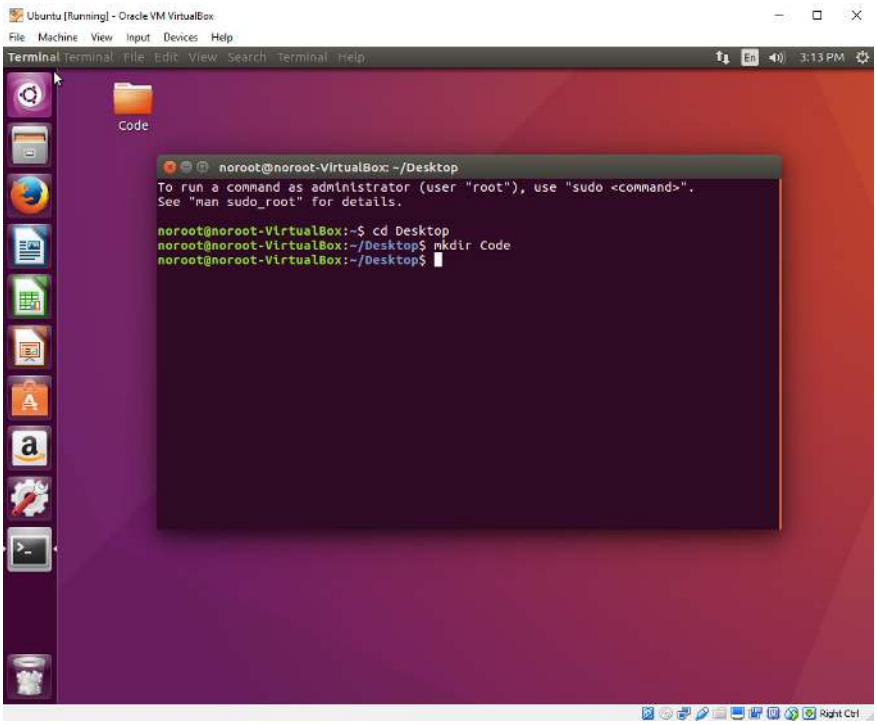
Congratulations! You have a working version of Linux!



Click on the top left icon and type terminal and double-click on the first Terminal icon with the >_ in the window.



You will see a Terminal icon at the bottom left of your screen. Right-click on it and select Lock to Launcher so that it will be available for you once you close the window.



In the terminal window type cd Desktop and press Enter. Then type mkdir Code and press enter. The first command moves you into the Desktop directory and the mkdir command creates a folder on the Desktop called Code so that we have a place to store our software applications that we create.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ sudo apt-get update
```

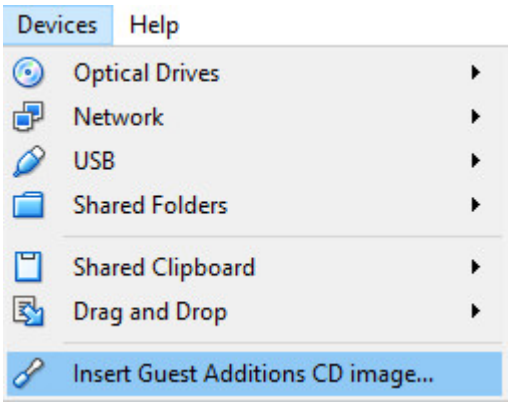
It is important you keep your version of Linux up to date. Every time you login, you should type the following commands. First, sudo apt-get update and press enter.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ sudo apt-get upgrade
```

Next you should then type sudo apt-get upgrade and press enter.

```
noroot@noroot-VirtualBox:~/Desktop$ sudo apt-get install gcc-multilib
```

In order to work with 32-bit Assembly examination, we need to install the gcc multilib package so that we can compile 32-bit versions of C code for examination. Type sudo apt-get install gcc-multilib and press enter.



Finally click on **Devices** and click **Insert Guest Additions CD Image…** in order to get a better working functionality out of your VM.

This has been a very long tutorial however necessary to get you a working copy of Linux so that we can continue with our future tutorials.

I look forward to seeing you all next week when we learn how to use the vim text editor to begin coding!

Part 18 - vim Text Editor

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Now that we have a working version of Linux, we need a text editor that we can work with in the terminal.

To begin, open your terminal and type:

```
noroot@noroot-VirtualBox:~$ cd ~
noroot@noroot-VirtualBox:~$ vi .vimrc
```

This will open up the vi text editor. The first thing you need to type is the letter 'i' to set the editor to insert mode so you may begin typing.

```
set number
set smartindent
set tabstop=4
set shiftwidth=4
set expandtab
```

After you a done typing, press the '**esc**' key and type '**:wq**' and press enter.

Congratulations! You created your first file! This is a one time file that we need to create in order to use our text editor they way we want it to perform.

The first line states **set number** which means we would like each file to show line numbers as this is essential for debugging code. The **set smartindent**, **set tabstop**, **set shiftwidth** and **set expandtab** statements set forth rules to properly format code and allow 4 spaces per tab indent which will help our code to look clean.

There are several commands you need to be aware of. Keep in mind, to go into command mode rather than insert mode you must press the '**esc**' key. Below are the most common commands:

j or down-arrow [move cursor down one line]

k or up-arrow [move cursor up one line]

h or left-arrow [move cursor left one character]

l or right-arrow [move cursor right one character]

0 [move cursor to the start of the current line]

$ [move cursor to the end of the current line]

b [move cursor back to the beginning of preceding word]

dd [deletes the line the cursor is on]

D [deletes from the cursor position to the end of the line]

yy [copies the current line]

p [puts the copied text after the cursor]

u [undo the last change to the file]

:w [save file]

:wq [save file and exit text editor]

:q! [quit text editor and do not save any changes]

You will be consistently moving between command mode '**esc**' and insert mode '**i**'. Remember that when you want to insert characters you need to be in insert mode and when you want to move the cursor other than moving to the next line, you need to be in command mode.

Now that we have vi configured, lets install vim which has some better functionality. Simply type:

```
noroot@noroot-VirtualBox:~/Desktop$ sudo apt-get install vim
```

Once that is installed instead of using vi we will now use vim.

I look forward to seeing you all next week when we talk about why it's important to learn Assembly Language.

Part 19 - Why Learn Assembly

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Why learn Assembly Language? Java is the most in-demand programming language and will get me a job immediately so why in the hell would I ever waste my damn time learning this archaic Assembly Language crap?

So many people ask me this question and it is true, Java is HOT and in the greatest demand and there is nothing wrong with learning Java however the threats that face society more than anything in this world, above everything else, is the Cyber Security threat. With that said, Java offers a great career path and I would encourage you to learn it however Java is not the only game in town.

Most malware is written in higher-level languages however most malware authors do not give the attackers their source code so they can properly deal with their crafted attack.

The hackers use a multitude of high-level languages and the demand for new professional Malware Analyst Reverse Engineers continue to grow daily.

When we examine malware, more than not we get only a compiled binary. The only thing we can do with a compiled binary is to break it down, instruction-by-instruction, in Assembly Language as EVERYTHING ultimately goes down to Assembly Language.

When someone says Assembly Language is a dinosaur I say to those people, lets have that conversation when your entire network is brought to its knees and you can't login to a single terminal or manipulate a single machine on your network. Lets talk about how useless Assembly Language is at that time.

Understanding Assembly Language allows one to open a debugger on an a running process. Each running program has a PID to which is a numerical value which designates a running program. If we open a running process or any bit of malware with a professional or open-source tool like GDB, we can see EXACTLY what is going on and then grab the EIP instruction pointer to go where we need it to go to have COMPLETE control over program flow.

Most malware is written, as I have stated, in a middle-level language and once compiled it can be read by the hardware or OS as it is not human-readable. In order for professional Cyber Security Engineers to understand this, they must learn to read, write and properly debug Assembly.

Assembly Language is low-level and has many more instructions than you would see in a higher-level application.

The prior 18 lessons in this tutorial series gave you the basics of x86 hardware. As I have stated in prior tutorials, we will focus on 32-bit Assembly debugging as most malware is going to try to affect as many systems as possible and although

there is 64-bit malware, 32-bit malware is significantly more destructive and dangerous and will be the focus of this series.

I look forward to seeing you all next week when we learn the basics of instruction code handling.

Part 20 - Instruction Code Handling

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

A CPU reads instruction codes that are stored in memory as each code set can contain one of more bytes of information that guide the processor to perform a very specific task. As each instruction code is read in from memory, any data needed for the instruction code is also stored and read into memory.

Keep in mind, memory that contain instruction codes are no different than the bytes that contain the data used by the CPU and special pointers are used to help the CPU keep track of where in memory data is and where instruction codes are stored.

A data pointer helps the CPU keep track of where the data area in memory starts which is the stack. When new data elements are placed in the stack, the stack pointer moves down in memory and as data is read from the stack the stack pointer moves up in memory. Please review Part 15 – Stack if you don't understand this concept.

The instruction pointer is used to help the CPU keep track of which instruction codes have already been processed and what code is to be processed next. Please review Part 12 – Instruction Pointer Register if you don't understand this concept.

Each and every instruction code must include an opcode that defines the basic function or task to be performed by the CPU to which opcodes are between 1 and 3 bytes in length and uniquely defines the function that is performed.

Lets examine a simple C program called test.c to get started.

```
1 int main(void) {
2     return 0;
3 }
```

All we are doing is creating a main function of type integer to which it has a void parameter and returning 0. All this program does is simply exit the OS.

Lets compile and run this program.

```
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o test test.c
```

Lets use the objdump tool to and find the main function within it.

```
noroot@noroot-VirtualBox:~/Desktop$ objdump -d -M intel test | grep main.: -A11
```

Here is a snippet of the results you would get by running the above command. Here are the contents of the main function. Keep in mind the below is in Intel syntax as we spoke about in the last tutorial.

```
080483db <main>:
 80483db:       55                      push   ebp
 80483dc:       89 e5                   mov    ebp,esp
 80483de:       b8 00 00 00 00          mov    eax,0x0
 80483e3:       5d                      pop    ebp
 80483e4:       c3                      ret
 80483e5:       66 90                   xchg   ax,ax
 80483e7:       66 90                   xchg   ax,ax
 80483e9:       66 90                   xchg   ax,ax
 80483eb:       66 90                   xchg   ax,ax
 80483ed:       66 90                   xchg   ax,ax
 80483ef:       90                      nop
```

On the far left we have the corresponding memory addresses. In the center we have the opcodes and finally on the right we have the corresponding assembly language in Intel syntax.

To keep this simple, lets examine memory address **80483de** where we see op codes **b8 00 00 00 00**. We can see that the **b8** opcode corresponds with the **mov eax, 0x0** instruction on the right. The next series of 00 00 00 00 represents 4 bytes of the value 0. We see **mov eax, 0x0** therefore the value of 0 is moved into eax therefore representing the above code. Keep in mind, the IA-32 platform uses what we call little-endian notation which means the lower-value bytes appear first in order when reading right to left.

I want to make sure you have this straight in your head so lets pretend the value above was:

mov eax, 0x1

In this scenario the corresponding opcode would be:

b8 01 00 00 00

If you are confused it is ok. Remember little-endian? Keep in mind eax is 32-bits wide therefore that is 4 bytes (8 bits = 1 byte). The values are listed in reverse order therefore we see the above representation.

I look forward to seeing you all next week when we dive into the details about how to compile a program.

Part 21 - How To Compile A Program

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's look again at last weeks C program and take a deeper look at how we turn that source code into an executable file.

```
1 int main(void) {
2     return 0;
3 }
```

To compile this program in C, we simply type:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -m32 -ggdb -o exit exit.c
```

This single step will create **exit.o** which is the binary object file and **exit** which is the binary executable file.

If we wanted to convert this C source code to Assembly, we need to use the GNU compiler in the below fashion. Lets start by running the below command in the terminal:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -S -m32 -O0 exit.c
```

Let's begin with the -S switch. The -S switch will create comparable AT&T Syntax Assembly source code. The -m32 will create a 32-bit executable and the -O0 will tell the compiler how much optimization to use when compiling the binary. That is the capital O and the numeric 0. Numeric 0 in that case means no optimization which means it is at the most human readable instruction set. If you were to substitute a 1, 2 or 3 the amount of optimization increases as the values go up.

```
 1     .file   "exit.c"
 2     .text
 3     .globl  main
 4     .type   main, @function
 5 main:
 6 .LFB0:
 7     .cfi_startproc
 8     pushl   %ebp
 9     .cfi_def_cfa_offset 8
10     .cfi_offset 5, -8
11     movl    %esp, %ebp
12     .cfi_def_cfa_register 5
13     movl    $0, %eax
14     popl    %ebp
15     .cfi_restore 5
16     .cfi_def_cfa 4, 4
17     ret
18     .cfi_endproc
19 .LFE0:
20     .size   main, .-main
21     .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.1) 5.4.0 20160609"
22     .section    .note.GNU-stack,"",@progbits
```

This step above creates exit.s which is the equivalent Assembly Language source code as we mentioned above.

We then need to compile the Assembly source code into a binary object file which will generate a **exit.o** file.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -m32 -c exit.s -o exit.o
```

Finally we need to use a linker to create the actual binary executable code from the binary object file which will create an executable called exit.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -m32 exit.o -o exit
```

Last week when we examined the executable file exit in a program called objdump, and examined the main area we saw the following below except this time we will use AT&T Assembly Language Syntax:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ objdump -d exit | grep main.: -A11
```

This command above will create the following output below:

```
080483db <main>:
 80483db:       55                      push   %ebp
 80483dc:       89 e5                   mov    %esp,%ebp
 80483de:       b8 00 00 00 00          mov    $0x0,%eax
 80483e3:       5d                      pop    %ebp
 80483e4:       c3                      ret
 80483e5:       66 90                   xchg   %ax,%ax
 80483e7:       66 90                   xchg   %ax,%ax
 80483e9:       66 90                   xchg   %ax,%ax
 80483eb:       66 90                   xchg   %ax,%ax
 80483ed:       66 90                   xchg   %ax,%ax
 80483ef:       90                      nop
```

Lets examine the code in the debugger. Let's start GDB which is the GNU debugger and first list the source code by typing l, then set a breakpoint on main and run the program. Finally we will disassemble and review the output below:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q exit
Reading symbols from exit...done.
(gdb) l
1       int main(void) {
2           return 0;
3       }
(gdb) b main
Breakpoint 1 at 0x80483de: file exit.c, line 2.
(gdb) r
Starting program: /home/noroot/Desktop/Code/exit

Breakpoint 1, main () at exit.c:2
2           return 0;
(gdb) disas
Dump of assembler code for function main:
   0x080483db <+0>:     push   %ebp
   0x080483dc <+1>:     mov    %esp,%ebp
=> 0x080483de <+3>:     mov    $0x0,%eax
   0x080483e3 <+8>:     pop    %ebp
   0x080483e4 <+9>:     ret
End of assembler dump.
```

In each of the three above examinations, you will essentially see the same set of instructions which we will take a deeper look as to what is exactly going on in future tutorials.

Throughout this tutorial series thus far we have been looking at Intel Syntax Assembly Language. We are going to turn our focus to AT&T Syntax as I have stated above as this is the natural syntax utilized in Linux with the GNU Assembler and GNU Debugger.

The biggest different you will see is that in AT&T Syntax, the source and destinations are reversed.

AT&T Syntax : **movl %esp, %ebp** [This means move esp into ebp.]

Intel Syntax : **mov esp, ebp** [This means move ebp into esp.]

You will also see some additional variances as AT&T uses additional variances which we will cover in a later tutorial.

If we wanted to create a pure Assembly Code program which does the same thing above we would type:

```
1 .section .data
2
3 .section .bss
4
5 .section .text
6     .globl _start
7
8 _start:
9     nop                             # debugging break point
10
11 exit:
12     movl $1, %eax                  # sys_exit
13     movl $0, %ebx                  # display 0 if normal status
14     int $0x80                      # call sys_exit
```

To compile this we would use the GAS Assembler and Linker:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ as --32 -gstabs -o exit__s.o exit__s.s
noroot@noroot-VirtualBox:~/Desktop/Code$ ld -m elf_i386 -o exit__s exit__s.o
noroot@noroot-VirtualBox:~/Desktop/Code$ ./exit
```

To run any executable in Linux you type ./ and the name of the binary executable. In this case we type ./exit and press return. When we do so, nothing happens. That is good as all we did was create a program that exited to the OS.

I look forward to seeing you all next week when we dive into more assembly code!

Part 22 - ASM Program 1 [Moving Immediate Data]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

I appreciate everyone being patient as it has taken 21 lessons to get to our first ASM program however very necessary background had to be covered in order to fully understand where we begin when developing assembly language.

We are going to create 32-bit assembly programs as most malware is written in 32-bit mode in order to attack the maximum amount of systems possible. Keep in mind even though most of us ALL have 64-bit operating systems, 32-bit programs can run on them.

For the most part we have been working with Intel syntax when it comes to assembly however I am going to focus on the native AT&T syntax going forward. It is very easy to convert back and forth between Intel and AT&T syntax as I have demonstrated in prior tutorials.

Every assembly language program is divided into three sections:

1)**Data Section**: This section is used for declaring initialized data or constants as this data does not ever change at runtime. You can declare constant values, buffer sizes, file names, etc.

2)**BSS Section**: This section is used for declaring uninitialized data or variables.

3)**Text Section**: This section is used for the actual code sections as it begins with a global _start which tells the kernel where execution begins.

Critical to any development is the use of comments. In the AT&T syntax we use use the # symbol to declare a comment as any data after that symbol on a respective line will be ignored by the compiler.

Keep in mind, assembly language statements are entered in one statement per line as you do not have to end the line with a semicolon like many other languages. The structure of a statement is as follows:

[label] mnemonic [operands] [comment]

A basic instruction has two parts of which the first one is the name of the instruction or the mnemonic which is executed and the second part is the operands or parameters of the command.

Our first program will demonstrate how to move immediate data to a register and immediate data to memory.

Lets open VIM and create a program called **moving_immediate_data.s** and type the following:

```
#moving_immediate_data:  mov immediate data between registers & memory


.section .data

.section .bss
    .lcomm buffer 1

.section .text
    .globl _start

_start:
    nop                             #used for debugging purposes

mov_immediate_data_to_register:
    movl $100, %eax                 #mov 100 into the EAX register
    movl $0x50, buffer              #mov 0x50 into buffer memory location

exit:
    movl $1, %eax                   #sys_exit system call
    movl $0, %ebx                   #exit code 0 successful execution
    int $0x80                       #call sys_exit
```

To compile you type:

as –32 -o moving_immediate_data.o moving_immediate_data.s

ld -m elf_i386 -o moving_immediate_data moving_immediate_data.o

To run you type:

./moving_immediate_data

I would like to show you what it would look like in Intel syntax as well. Before we examine this part you will need to type **sudo apt-get install nasm** in a command prompt which will install the Netwide Assembler:

```
;moving_immediate_data:  mov immediate data between registers & memory


section .data

section .bss
    buffer resb 1

section .text
    global _start

_start:
    nop                             ;used for debugging purposes

mov_immediate_data_to_register:
    mov eax, 100                    ;mov 100 into the EAX register
    mov byte[buffer], 0x50          ;mov 0x50 into buffer memory location

exit:
    mov eax, 1                      ;sys_exit system call
    mov ebx, 0                      ;exit code 0 successful execution
    int 0x80                        ;call sys_exit
```

To compile you type:

nasm -f elf32 moving_immediate_data.asm

ld -m elf_i386 -o moving_immediate_data moving_immediate_data.o

To run you type:

./moving_immediate_data

Ok what the heck! There is no output! That is correct and you did not do anything wrong. Many of our programs will not actually do anything as they are not much more than sandbox programs that we will use in GDB for analysis and manipulation.

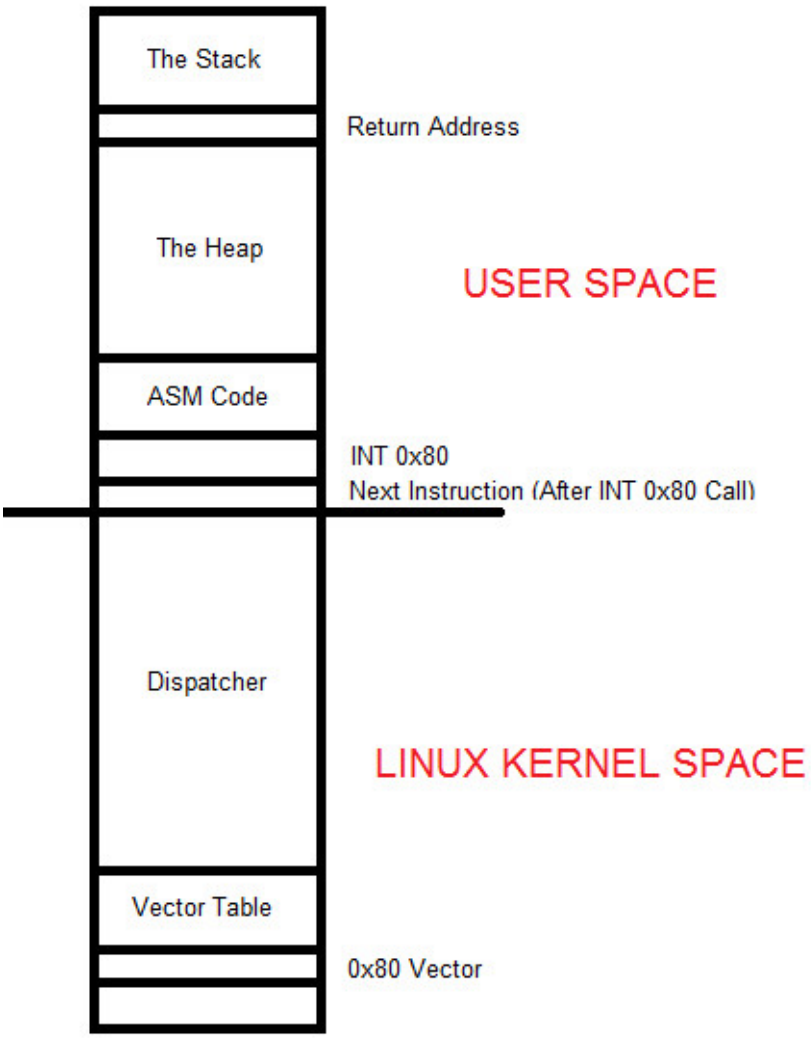
Next week we will dive into the GNU GDB debugger and see what is going on under the hood.

I want to take some time and discuss the code at line 20 – 22 in the AT&T version and the Intel Syntax version as well. This set of instructions takes advantage of what we call a software interrupt. On line 20 in the AT&T Syntax, we **movl $1, %eax** meaning we move the decimal value of 1 into eax which specifies the sys_exit call which will properly terminate program execution back to Linux so that there is no segmentation fault. On line 21, we **movl $0, %ebx** which moves 0 into ebx to show that the program successfully executed and finally we see int **$0x80**.

Line 20 and 21 set up the software interrupt which we call on line 22 with the instruction int **$0x80.** Let's dive into this a little deeper.

In Linux, there are two distinct areas of memory. At the very bottom of memory in any program execution we have the Kernel Space which is made up of the Dispatcher section and the Vector Table.

At the very top of memory in any program execution we have the User Space which is made up of The Stack, The Heap and finally your code all of which can be illustrated in the below diagram:



When we load the values as we demonstrated above and call INT 0x80, the very next instruction's address in the User Space, ASM Code section which is your code, is placed into the Return Address area in The Stack. This is critical so that when INT 0x80 does its work, it can properly know what instruction is to be carried out next to ensure proper and sequential program execution.

Keep in mind in modern versions of Linux, we are utilizing Protected Mode which means you do NOT have access to the Linux Kernel Space. Everything under the long line that runs in the middle of the diagram above represents the Linux Kernel Space.

The natural question is why can't we access this? The answer is very simple, Linux will NOT allow your code to access operating system internals as that would be very dangerous as any Malware could manipulate those components of the OS to track all sorts of things such as user keystrokes, activities and the like.

In addition, modern Linux OS architecture changes the address of these key components constantly as new software is installed and removed in addition to system patches and upgrades. This is the cornerstone of Protected Mode operating systems.

The way that we have our code communicate with the Linux Kernel is through the use of a kernel servies call gate which is a protected gateway between User Space where your program is running and Kernel Space which is implemented through the Linux Software Interrupt of 0x80.

At the very, very bottom of memory where segment 0, offset 0 exists is a lookup table with 256 entries. Every entry is a memory address including segment and offset portions which comprise of 4 bytes per entry as the first 1,024 bytes are reserved for this table and NO OTHER CODE can be manipulated there. Each address is called an interrupt vector which comprises the whole called the interrupt vector table where every vector has a number from 0 to 255 to which vector 0 starts off occupying bytes 0 to 3. This continues with vector 1 which contains 4 to 7, etc.

Keep in mind, none of these addresses are part of permanent memory. What is static is vector 0x80 which points to the services dispatcher which point to Linux kernel service routines.

When the return address is popped off the stack returns to the next instruction, the instruction is called the Interrupt Return or IRET which completes the execution of program flow.

Take some time and look at the entire table of system calls by opening up a terminal and typing:

cat /usr/include/asm/unistd_32.h

Below is a snapshot of just a few of them. As you can see the exit 1 represents the sys_exit that we utilized in our above code.

```
noroot@noroot-VirtualBox:/usr/include/asm$ cat /usr/include/asm/unistd_32.h
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
```

Starting with this lesson we will take a 3-step approach:

1)**Program**

2)**Debug**

3)**Hack**

Each week we will start with a program like you see here, the following week we will take it into GDB and examine what exactly is going on at the assembly level and finally in the third series of each week we will hack the data in GDB to change it to whatever we want demonstrating the ability to control program flow which includes learning how to hack malware to a point where it is not a threat.

We will not necessarily look at malware directly as I would rather focus on the topics of assembly language programs that will give you the tools and understanding so that ANY program can be debugged and manipulated to your liking. That is the purpose of these tutorials.

The information you will learn in this tutorial series can be used with high-level GUI debuggers like IDA Pro as well however I will focus only on the GNU GDB debugger.

I look forward to seeing you all next week when we dive into creating our first assembly debug!

Part 23 - ASM Debugging 1 [Moving Immediate Data]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's begin by loading the binary into GDB.

To load into GDB type:

gdb -q moving_immediate_dat

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_immediate_data
Reading symbols from moving_immediate_data...(no debugging symbols found)...done
.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_immediate_data

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:     nop
End of assembler dump.
```

Let's first set a breakpoint on start by typing b _start.

We can then run the program by typing **r**.

To then begin disassembly, we simply type **disas**.

We coded a **nop** which means no operation or 0x90 from an OPCODE perspective for proper debugging purposes which the breakpoint properly hit. This is good practice when creating assembly programs.

```
(gdb) si
0x08048075 in mov_immediate_data_to_register ()
(gdb) disas
Dump of assembler code for function mov_immediate_data_to_register:
=> 0x08048075 <+0>:     mov    $0x64,%eax
   0x0804807a <+5>:     movl   $0x50,0x8049090
End of assembler dump.
```

The native syntax as I have stated many times before is AT&T syntax which you see above. I painfully go back and forth deliberately so that you have comfort in each however going forward I will be sticking to the AT&T syntax however wanted to show you a few examples of both. I will state again that if you ever want to see Intel syntax simply type set-disassembly-flavor intel and you will have what you are looking for.

We first use the command si which means step-into to advance to the next instruction. What we see here at **_start+0** is you are moving the hex value of **0x64** into **EAX**. This is simply moving decimal **100** or as the computer sees it, hex **0x64** into **EAX** which demonstrates moving an immediate value into a register.

```
(gdb) si
0x0804807a in mov_immediate_data_to_register ()
(gdb) i r
eax            0x64     100
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xffffd040       0xffffd040
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804807a        0x804807a <mov_immediate_data_to_register+5>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
```

We step-into again and then use the command i r which keep in mind has a space between them to give us information on the state of the CPU registers. We can see EAX now has the value of 0x64 hex or 100 decimal.

```
(gdb) si
0x08048084 in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x08048084 <+0>:     mov    $0x1,%eax
   0x08048089 <+5>:     mov    $0x0,%ebx
   0x0804808e <+10>:    int    $0x80
End of assembler dump.
(gdb) print /x buffer
$1 = 0x50
```

After we step-into again and do a **disas**, we see that we have then moved the value of **0x50** into the **buffer** label as can refer back to the source code from last week to see.

When dealing with non-register data, we can use the print command above as we type **print /x buffer** and it clearly shows us that the value inside buffer is **0x50**. The **/x** designation means show us the value in hex.

```
(gdb) x/xb 0x8049090
0x8049090 <buffer>:     0x50
```

Consequently you can review slide 2 of this tutorial above you see at **_start+5** the immediate value of **0x50** loaded into the **buffer** label or in this case the address of **buffer** which is **0x8049090** and we can examine it by using the examine instruction by typing **x/xb 0x8049090** which shows us one hex byte at that location which yields **0x50.**

We will be doing this with every program example so that we can dive into the debugging process. If there are any questions, please leave them below in the comments.

I look forward to seeing you all next week when we dive into creating our first assembly hack!

Part 24 - ASM Hacking 1 [Moving Immediate Data]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's begin by loading the binary into GDB.

To load into GDB type:

gdb -q moving_immediate_data

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_immediate_data
Reading symbols from moving_immediate_data...(no debugging symbols found)...done
.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_immediate_data

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:     nop
End of assembler dump.
```

Let's first set a breakpoint on start by typing **b _start**.

We can then run the program by typing **r**.

To then begin disassembly, we simply type **disas**.

We coded a **nop** which means no operation or **0x90** from an OPCODE perspective for proper debugging purposes which the breakpoint properly hit. This is good practice when creating assembly programs.

```
(gdb) si
0x08048075 in mov_immediate_data_to_register ()
(gdb) disas
Dump of assembler code for function mov_immediate_data_to_register:
=> 0x08048075 <+0>:     mov    $0x64,%eax
   0x0804807a <+5>:     movl   $0x50,0x8049090
End of assembler dump.
```

Lets have some fun! At this point lets **si** once and do an **i r** to see that **0x64** has in fact been moved into **EAX**.

```
(gdb) si
0x08048075 in mov_immediate_data_to_register ()
(gdb) disas
Dump of assembler code for function mov_immediate_data_to_register:
=> 0x08048075 <+0>:     mov    $0x64,%eax
   0x0804807a <+5>:     movl   $0x50,0x8049090
End of assembler dump.
(gdb) si
0x0804807a in mov_immediate_data_to_register ()
(gdb) i r
eax            0x64     100
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xffffd040       0xffffd040
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804807a        0x804807a <mov_immediate_data_to_register+5>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

We can see **EAX** has the value of **0x64** or **100** decimal. Lets HACK that value now by setting **EAX** to say something like **0x66** by typing **set $eax = 0x66**.

```
(gdb) set $eax = 0x66
(gdb) i r
eax            0x66     102
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xffffd040       0xffffd040
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804807a        0x804807a <mov_immediate_data_to_register+5>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

BAM! There we go! You can see the ULTIMATE power of assembly here! We just hacked the value from **0x64** to **0x66** or **100** to **102** decimal. This is a trivial example however you can clearly see when you learn to master these concepts you develop a greater power over the computer. With each program that we create, we will have a very simple lesson like this where we will hijack at least one portion of the code so we can not only see how the program is created and debugged but how we can manipulate it to whatever we want.

I look forward to seeing you all next week when we dive into creating our second assembly program!

Part 25 - ASM Program 2 [Moving Data Between Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our second program we will demonstrate how we can move data between registers. Moving data from one register to another is the fastest way to move data. It is always advisable to keep data between registers as much as can be engineered for speed.

Specifically we will move the value in EDX into EAX. We will initialize this program with a simple immediate value of 22 decimal which will go into EDX and ultimately into EAX.

```
1  #moving_data_between_registers:  mov data between registers
2
3
4  .section .data
5
6  .section .text
7      .globl _start
8
9  _start:
10     nop                          #used for debugging purposes
11
12     movl $22, %edx               #mov immediate value into EDX
13
14 mov_data_between_registers:
15     movl %edx, %eax              #mov the value in EDX into EAX
16
17 exit:
18     movl $1, %eax                #sys_exit system call
19     movl $0, %ebx                #exit code 0 successful execution
20     int $0x80                    #call sys_exit
```

Keep in mind you can only move similar registers between each other. We know that EAX and EDX are 32-bit registers. We know that each of these registers can be accessed by their 16-bit values as ax and dx respectively. You can't move a 32-bit value into a 16-bit value and vice-versa.

I look forward to seeing you all next week when we dive into debugging our second assembly program!

Part 26 - ASM Debugging 2 [Moving Data Between Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's debug the second program below:

```
 1 #moving_data_between_registers:  mov data between registers
 2
 3
 4 .section .data
 5
 6 .section .text
 7     .globl _start
 8
 9 _start:
10     nop                         #used for debugging purposes
11
12     movl $22, %edx              #mov immediate value into EDX
13
14 mov_data_between_registers:
15     movl %edx, %eax             #mov the value in EDX into EAX
16
17 exit:
18     movl $1, %eax               #sys_exit system call
19     movl $0, %ebx               #exit code 0 successful execution
20     int $0x80                   #call sys_exit
```

Lets fire up GDB and break on _start, run the binary and disas:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers
Reading symbols from moving_data_between_registers...(no debugging symbols found
)...done.
(gdb) b _start
Breakpoint 1 at 0x8048054
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers

Breakpoint 1, 0x08048054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048054 <+0>:     nop
   0x08048055 <+1>:     mov    $0x16,%edx
End of assembler dump.
(gdb)
Dump of assembler code for function _start:
=> 0x08048054 <+0>:     nop
   0x08048055 <+1>:     mov    $0x16,%edx
```

Now lets **si** twice and **i r**:

```
(gdb) i r
eax            0x0      0
ecx            0x0      0
edx            0x16     22
ebx            0x0      0
esp            0xffffd030       0xffffd030
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804805a        0x804805a <mov_data_between_registers>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

As we can see the value of **0x16** or **22** decimal did move into EDX successfully.

Now lets **si** again.

```
(gdb) i r
eax            0x16     22
ecx            0x0      0
edx            0x16     22
ebx            0x0      0
esp            0xffffd030       0xffffd030
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804805c        0x804805c <exit>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

As you can see we have successfully moved EDX into EAX.

I look forward to seeing you all next week when we dive into hacking our second assembly program!

Part 27 - ASM Hacking 2 [Moving Data Between Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's hack the second program below:

```
#moving_data_between_registers:  mov data between registers


.section .data

.section .text
    .globl _start

_start:
    nop                             #used for debugging purposes

    movl $22, %edx                  #mov immediate value into EDX

mov_data_between_registers:
    movl %edx, %eax                 #mov the value in EDX into EAX

exit:
    movl $1, %eax                   #sys_exit system call
    movl $0, %ebx                   #exit code 0 successful execution
    int $0x80                       #call sys_exit
```

Lets fire up GDB and break on _start, run the binary and disas:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers
Reading symbols from moving_data_between_registers...(no debugging symbols found
)...done.
(gdb) b _start
Breakpoint 1 at 0x8048054
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers

Breakpoint 1, 0x08048054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048054 <+0>:     nop
   0x08048055 <+1>:     mov    $0x16,%edx
End of assembler dump.
(gdb)
Dump of assembler code for function _start:
=> 0x08048054 <+0>:     nop
   0x08048055 <+1>:     mov    $0x16,%edx
```

Now lets **si** twice and **i r**:

```
(gdb) si
0x08048055 in _start ()
(gdb) si
0x0804805a in mov_data_between_registers ()
(gdb) i r
eax            0x0      0
ecx            0x0      0
edx            0x16     22
ebx            0x0      0
esp            0xffffd030       0xffffd030
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804805a        0x804805a <mov_data_between_registers>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

As we can see the value of **0x16** or **22** decimal did move into EDX successfully. This is what we did in the last lesson however here we are going to hack that value to something else.

We can **set $edx = 0x19** for example:

```
(gdb) set $edx = 0x19
(gdb) i r
eax            0x0      0
ecx            0x0      0
edx            0x19     25
ebx            0x0      0
esp            0xffffd030       0xffffd030
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804805a        0x804805a <mov_data_between_registers>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

As you can see we easily hacked the value of **EDX** to **0x19** or **25** decimal.

Hopefully you see some very simple patterns now that we are diving into very simple assembly language programs. The key is to understand how to manipulate values and instructions so that you have complete control over the binary.

We are going to continue to move at a snails pace throughout the rest of this tutorial as my goal is to give everyone very small bite-size examples of how to understand x86 assembly.

I look forward to seeing you all next week when we dive into writing our third assembly program!

Part 28 - ASM Program 3 [Moving Data Between Memory And Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our third program we will demonstrate how we can move data between memory and registers.

```
1  #moving_data_between_memory_and_registers:  mov data between mem and regs
2
3
4  .section .data
5      constant:
6          .int 10
7
8  .section .text
9      .globl _start
10
11 _start:
12     nop                              #used for debugging purposes
13
14 mov_immediate_data_between_memory_and_registers:
15     movl constant, %ecx              #mov constant value into EAX register
16
17 exit:
18     movl $1, %eax                    #sys_exit system call
19     movl $0, %ebx                    #exit code 0 successful execution
20     int $0x80                        #call sys_exit
```

Specifically we will move the value of inside the constant integer of 10 decimal into ECX.

Keep in mind to assemble we type:

as –32 -o moving_data_between_memory_and_registers.o moving_data_between_memory_and_registers.s

To link the object file we type:

ld -m elf_i386 -o moving_data_between_memory_and_registers moving_data_between_memory_and_registers.o

I look forward to seeing you all next week when we dive into debugging our third assembly program!

Part 29 - ASM Debugging 3 [Moving Data Between Memory And Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's debug!

```
1 #moving_data_between_memory_and_registers:  mov data between mem and regs
2
3
4 .section .data
5     constant:
6         .int 10
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                             #used for debugging purposes
13
14 mov_immediate_data_between_memory_and_registers:
15     movl constant, %ecx             #mov constant value into EAX register
16
17 exit:
18     movl $1, %eax                   #sys_exit system call
19     movl $0, %ebx                   #exit code 0 successful execution
20     int $0x80                       #call sys_exit
```

Specifically we will move the value of inside the constant integer of 10 decimal into ECX.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_memory_and_r
egisters
Reading symbols from moving_data_between_memory_and_registers...(no debugging sy
mbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_memory_and_regis
ters

Breakpoint 1, 0x08048074 in _start ()
```

We open GDB in quiet mode and break on _start and run by following the commands above.

```
(gdb) i r
eax            0x0        0
ecx            0x0        0
edx            0x0        0
ebx            0x0        0
esp            0xffffd020         0xffffd020
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x8048074          0x8048074 <_start>
eflags         0x202      [ IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0        0
gs             0x0        0
(gdb)
```

As we can see when we info registers the value of ECX is 0.

```
(gdb) si
0x08048075 in mov_immediate_data_between_memory_and_registers ()
(gdb) si
0x0804807b in exit ()
(gdb) i r
eax            0x0      0
ecx            0xa      10
edx            0x0      0
ebx            0x0      0
esp            0xffffd020       0xffffd020
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804807b        0x804807b <exit>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

After we step into twice, we now see the value of ECX as 10 decimal of 0xa hex.

I look forward to seeing you all next week when we dive into hacking our third assembly program!

Part 30 - ASM Hacking 3 [Moving Data Between Memory And Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's hack!

```
#moving_data_between_memory_and_registers:  mov data between mem and regs


.section .data
    constant:
        .int 10

.section .text
    .globl _start

_start:
    nop                             #used for debugging purposes

mov_immediate_data_between_memory_and_registers:
    movl constant, %ecx             #mov constant value into EAX register

exit:
    movl $1, %eax                   #sys_exit system call
    movl $0, %ebx                   #exit code 0 successful execution
    int $0x80                       #call sys_exit
```

Specifically we will move the value of inside the constant integer of 10 decimal into ECX like before.

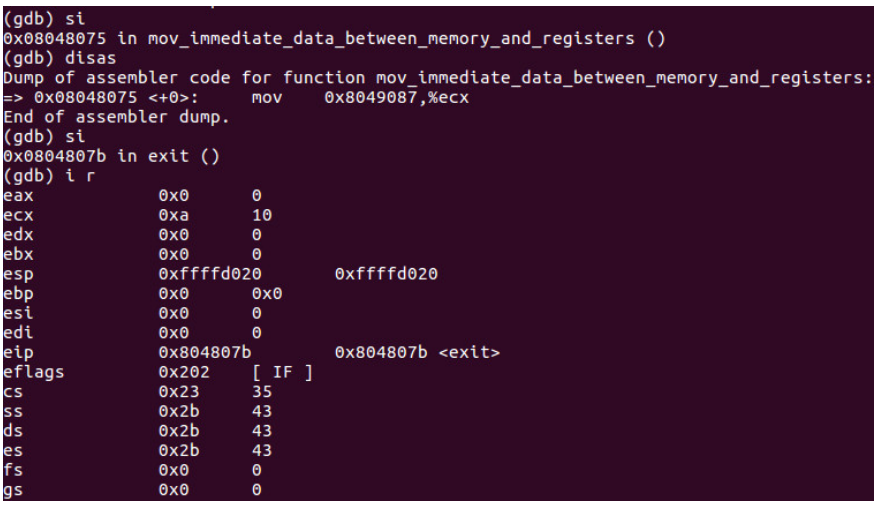
```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_memory_and_r
egisters
Reading symbols from moving_data_between_memory_and_registers...(no debugging sy
mbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_memory_and_regis
ters

Breakpoint 1, 0x08048074 in _start ()
```

We open GDB in quiet mode and break on _start and run by following the commands above.

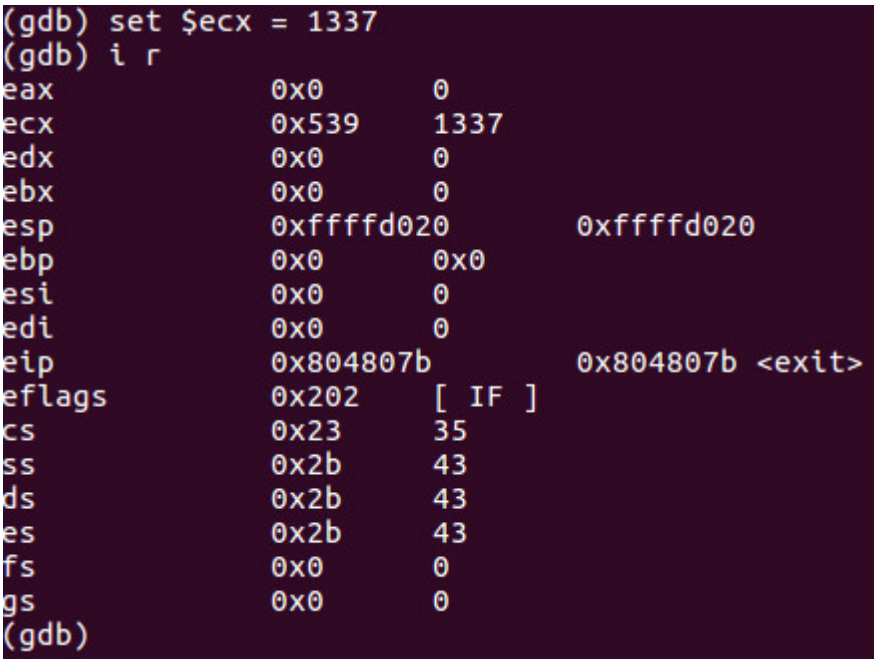
```
(gdb) i r
eax            0x0        0
ecx            0x0        0
edx            0x0        0
ebx            0x0        0
esp            0xffffd020         0xffffd020
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x8048074          0x8048074 <_start>
eflags         0x202      [ IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0        0
gs             0x0        0
(gdb)
```

As we can see when we info registers the value of ECX is 0. Let's do a si and another si.



As you can see the value of ECX is 10 decimal or 0xa hex as it was in the prior lesson now lets hack that value to something else.

Let's **set $ecx = 1337** and do an i r.



As you can clearly see we have hacked the value of ECX to 0x539 hex or 1337 decimal.

As I have stated throughout this series. Each of these lessons are very bite-sized examples so that you get the hard muscle memory on how to hack through a variety of situations so that you ultimately have a complete mastery of processor control.

I look forward to seeing you all next week when we dive into creating our fourth assembly program!

Part 31 - ASM Program 4 [Moving Data Between Registers And Memory]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our fourth program we will demonstrate how we can move data between registers and memory.

```
#moving_data_between_registers_and_memory:  mov data between regs and mem


.section .data
    constant:
        .int 10

.section .text
    .globl _start

_start:
    nop                             #used for debugging purposes

mov_immediate_data_between_registers_and_memory:
    movl $777, %eax                 #mov immediate value 777 to eax
    movl %eax, constant             #mov eax value into constant mem value

exit:
    movl $1, %eax                   #sys_exit system call
    movl $0, %ebx                   #exit code 0 successful execution
    int $0x80                       #call sys_exit
```

Specifically we will move the immediate value of 777 decimal into EAX. We then move that value stored in EAX into the constant value in memory which initially had the value of 10 decimal at runtime. Keep in mind we could have called the value anything however I called it constant as it was set up as a constant in the .data section.

You can clearly see it can be manipulated so it is NOT a constant. I chose constant deliberately as if it was in pure form the value would stay 10 decimal or 0xa hex.

This code is purely an academic exercise as variable data normally would be set up under the .bss section however I wanted to demonstrate that the above is possible to show the absolute flexibility of assembly language.

Keep in mind to assemble we type:

as –32 -o moving_data_between_registers_and_memory.o moving_data_between_registers_and_memory.s

To link the object file we type:

ld -m elf_i386 -o moving_data_between_registers_and_memory moving_data_between_registers_and_memory.o

I look forward to seeing you all next week when we dive into debugging our fourth assembly program!

Part 32 - ASM Debugging 4 [Moving Data Between Registers And Memory]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our fourth program we will demonstrate how we can move data between registers and memory.

```
 1 #moving_data_between_registers_and_memory:  mov data between regs and mem
 2
 3
 4 .section .data
 5     constant:
 6         .int 10
 7
 8 .section .text
 9     .globl _start
10
11 _start:
12     nop                             #used for debugging purposes
13
14 mov_immediate_data_between_registers_and_memory:
15     movl $777, %eax                 #mov immediate value 777 to eax
16     movl %eax, constant             #mov eax value into constant mem value
17
18 exit:
19     movl $1, %eax                   #sys_exit system call
20     movl $0, %ebx                   #exit code 0 successful execution
21     int $0x80                       #call sys_exit
```

Specifically we will move the immediate value of 777 decimal into EAX. We then move that value stored in EAX into the constant value in memory which initially had the value of 10 decimal at runtime. Keep in mind we could have called the value anything however I called it constant as it was set up as a constant in the .data section.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers_an
d_memory
Reading symbols from moving_data_between_registers_and_memory...(no debugging sy
mbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers_and_me
mory

Breakpoint 1, 0x08048074 in _start ()
(gdb) si
0x08048075 in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807a in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807f in exit ()
(gdb) print constant
$1 = 777
(gdb)
```

As you can see above we go into GDB and clearly see that the value of constant has been replaced with 777 decimal where in the code it was clearly set at 10 decimal in line 6 of the code at the beginning of this tutorial.

We can clearly see that in line 16 of the code the value of 777 decimal was successfully moved into EAX and into the memory value of constant.

I look forward to seeing you all next week when we dive into hacking our fourth assembly program!

Part 33 - ASM Hacking 4 [Moving Data Between Registers And Memory]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine the source code.

```
1  #moving_data_between_registers_and_memory:  mov data between regs and mem
2
3
4  .section .data
5      constant:
6          .int 10
7
8  .section .text
9      .globl _start
10
11 _start:
12     nop                             #used for debugging purposes
13
14 mov_immediate_data_between_registers_and_memory:
15     movl $777, %eax                 #mov immediate value 777 to eax
16     movl %eax, constant             #mov eax value into constant mem value
17
18 exit:
19     movl $1, %eax                   #sys_exit system call
20     movl $0, %ebx                   #exit code 0 successful execution
21     int $0x80                       #call sys_exit
```

We again can see above that we will move the immediate value of 777 decimal into EAX. We then move that value stored in EAX into the constant value in memory which initially had the value of 10 decimal at runtime. Keep in mind we could have called the value anything however I called it constant as it was set up as a constant in the .data section.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers_an
d_memory
Reading symbols from moving_data_between_registers_and_memory...(no debugging sy
mbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers_and_me
mory

Breakpoint 1, 0x08048074 in _start ()
(gdb) si
0x08048075 in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807a in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807f in exit ()
(gdb) print constant
$1 = 777
(gdb)
```

As you can see above we go into GDB and clearly see that the value of constant has been replaced with 777 decimal where in the code it was clearly set at 10 decimal in line 6 of the code at the beginning of this tutorial.

We can clearly see that in line 16 of the code the value of 777 decimal was successfully moved into EAX and into the memory value of constant.

Now lets hack this thing!

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers_an
d_memory
Reading symbols from moving_data_between_registers_and_memory...(no debugging sy
mbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers_and_me
mory

Breakpoint 1, 0x08048074 in _start ()
(gdb) si
0x08048075 in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807a in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807f in exit ()
(gdb) print constant
$1 = 777
(gdb) set constant = 666
(gdb) print constant
```

We took the very steps as we did last time with the debugging lesson. Here we hack the value of constant to which we hack the value from 777 to 666.

I look forward to seeing you all next week when we dive into creating our fifth assembly program!

Part 34 - ASM Program 5 [Indirect Addressing With Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our fifth program we will demonstrate how we can manipulate indirect addressing with registers.

```
1  #indirect_addressing_with_registers:  accessing data in mem using pointers
2
3
4  .section .data
5      constants:
6          .int 5, 8, 17, 44, 50, 52, 60, 65, 70, 77, 80
7
8  .section .text
9      .globl _start
10
11 _start:
12     nop                             #used for debugging purposes
13
14 indirect_addressing_with_registers:
15     movl constants, %eax            #mov constants mem value into eax
16     movl $constants, %edi           #mov mem addr into edi
17     movl $25, 4(%edi)               #mov immediate val 4b after edi ptr
18     movl $1, %edi                   #load 2nd index constants label
19     movl constants(, %edi, 4), %ebx #mov above value 4 bytes from constants
20
21 exit:
22     movl $1, %eax                   #sys_exit system call
23     movl $0, %ebx                   #exit code 0 successful execution
24     int $0x80                       #call sys_exit
```

We can place more than one value in memory as indicated above. In the past, our memory location contained one single value. In the above as you can see the value of constants contains 11 separate values.

This creates a sequential series of data values placed in memory. Each data value occupies one unit of memory which is an integer or 4 bytes.

We must use an index system to determine these values as what we have created above is an array.

We will utilize the indexed memory mode where the memory address is determined by a base address, an offset address to add to the base address and the size of the data element, in our case an integer of 4 bytes and an index to determine which data element to select.

Keep in mind an array starts with index 0. Therefore in the above code we see 1 moving into edi which is the 2nd index which ultimately goes into ebx.

We will dive deeper into this in the next lesson we debug however I want you to take some time to study the code above and get a good grasp of what is going on.

Keep in mind to assemble we type:

as –32 -o indirect_addressing_with_registers.o indirect_addressing_with_registers.s

To link the object file we type:

ld -m elf_i386 -o indirect_addressing_with_registers indirect_addressing_with_registers.o

I look forward to seeing you all next week when we dive into debugging our fifth assembly program!

Part 35 - ASM Debugging 5 [Indirect Addressing With Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our fifth program we demonstrated how we can manipulate indirect addressing with registers.

```
#indirect_addressing_with_registers:  accessing data in mem using pointers


.section .data
    constants:
        .int 5, 8, 17, 44, 50, 52, 60, 65, 70, 77, 80

.section .text
    .globl _start

_start:
    nop                             #used for debugging purposes

indirect_addressing_with_registers:
    movl constants, %eax            #mov constants mem value into eax
    movl $constants, %edi           #mov mem addr into edi
    movl $25, 4(%edi)               #mov immediate val 4b after edi ptr
    movl $1, %edi                   #load 2nd index constants label
    movl constants(, %edi, 4), %ebx #mov above value 4 bytes from constants

exit:
    movl $1, %eax                   #sys_exit system call
    movl $0, %ebx                   #exit code 0 successful execution
    int $0x80                       #call sys_exit
```

I want to start by addressing the question of why I use AT&T syntax. In previous lessons I provided many ways to easily convert back and forth between AT&T syntax and Intel syntax.

I deliberately choose this path so that it forces you to be comfortable with the most complex flavor of x86. If you are confused with this syntax please review the prior lessons as I go through the differences between both.

Let's recap. We will use objdump to take a compiled binary such as the one above that we compiled in our last lesson and show how we can view it's Intel source code.

objdump -d -M intel indirect_addressing_with_registers | grep _start.: -A24

```
noroot@noroot-gamma:~/Desktop/Code$ objdump -d -M intel indirect_addressing_with
_registers | grep _start.: -A24
08048074 <_start>:
 8048074:       90                      nop

08048075 <indirect_addressing_with_registers>:
 8048075:       a1 9e 90 04 08          mov    eax,ds:0x804909e
 804807a:       bf 9e 90 04 08          mov    edi,0x804909e
 804807f:       c7 47 04 19 00 00 00    mov    DWORD PTR [edi+0x4],0x19
 8048086:       bf 01 00 00 00          mov    edi,0x1
 804808b:       8b 1c bd 9e 90 04 08    mov    ebx,DWORD PTR [edi*4+0x804909e]

08048092 <exit>:
 8048092:       b8 01 00 00 00          mov    eax,0x1
 8048097:       bb 00 00 00 00          mov    ebx,0x0
 804809c:       cd 80                   int    0x80
```

Now back to our regularly scheduled program.

Let's load the binary into GDB and break on _start, step a few steps and examine 6 of the 11 values inside the constants label.

```
noroot@noroot-gamma:~/Desktop/Code$ gdb -q indirect_addressing_with_registers
Reading symbols from indirect_addressing_with_registers...(no debugging symbols
found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/indirect_addressing_with_registers

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:     nop
End of assembler dump.
(gdb) si
0x08048075 in indirect_addressing_with_registers ()
(gdb) disas
Dump of assembler code for function indirect_addressing_with_registers:
=> 0x08048075 <+0>:     mov    0x804909e,%eax
   0x0804807a <+5>:     mov    $0x804909e,%edi
   0x0804807f <+10>:    movl   $0x19,0x4(%edi)
   0x08048086 <+17>:    mov    $0x1,%edi
   0x0804808b <+22>:    mov    0x804909e(,%edi,4),%ebx
End of assembler dump.
(gdb) si
0x0804807a in indirect_addressing_with_registers ()
(gdb) x/6 &constants
0x804909e:      5       8       17      44
0x80490ae:      50      52
```

We then move the memory address of the constants label into edi and move the immediate value of 25 decimal into the second index of our array. This is in essence a source code hack as we are changing the original value of 8 to 25.

If you examine the source code you see line 18 where we load the value of 1 into edi. Keep in mind this is the second value as arrays are 0 based.

```
(gdb) si
0x0804807f in indirect_addressing_with_registers ()
(gdb) disas
Dump of assembler code for function indirect_addressing_with_registers:
   0x08048075 <+0>:     mov    0x804909e,%eax
   0x0804807a <+5>:     mov    $0x804909e,%edi
=> 0x0804807f <+10>:    movl   $0x19,0x4(%edi)
   0x08048086 <+17>:    mov    $0x1,%edi
   0x0804808b <+22>:    mov    0x804909e(,%edi,4),%ebx
End of assembler dump.
(gdb) si
0x08048086 in indirect_addressing_with_registers ()
(gdb) x/6 &constants
0x804909e:      5       25      17      44
0x80490ae:      50      52
```

You can see we changed the value of 8 decimal into 25 as explained.

This is our first introduction to arrays in assembly language. It is critical that you understand how they work as you may someday be a Malware Analyst or Reverse Engineer looking at the compiled binary of any number of higher-level program compiled arrays.

In our next lesson we will manually hack one of the values in GDB. Keep in mind, we will have to overwrite the contents inside an actual memory address with an immediate value. The fun is only beginning!

I look forward to seeing you all next week when we dive into hacking our fifth assembly program!

Part 36 - ASM Hacking 5 [Indirect Addressing With Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's reexamine the source once more.

```
1  #indirect_addressing_with_registers:  accessing data in mem using pointers
2
3
4  .section .data
5      constants:
6          .int 5, 8, 17, 44, 50, 52, 60, 65, 70, 77, 80
7
8  .section .text
9      .globl _start
10
11 _start:
12     nop                             #used for debugging purposes
13
14 indirect_addressing_with_registers:
15     movl constants, %eax            #mov constants mem value into eax
16     movl $constants, %edi           #mov mem addr into edi
17     movl $25, 4(%edi)               #mov immediate val 4b after edi ptr
18     movl $1, %edi                   #load 2nd index constants label
19     movl constants(, %edi, 4), %ebx #mov above value 4 bytes from constants
20
21 exit:
22     movl $1, %eax                   #sys_exit system call
23     movl $0, %ebx                   #exit code 0 successful execution
24     int $0x80                       #call sys_exit
```

Let's once again load the binary into GDB and break on _start.

```
noroot@noroot-gamma:~/Desktop/Code$ gdb -q indirect_addressing_with_registers
Reading symbols from indirect_addressing_with_registers...(no debugging symbols
found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/indirect_addressing_with_registers

Breakpoint 1, 0x08048074 in _start ()
(gdb) print *0x804909e
$1 = 5
(gdb) print *0x80490a2
$2 = 8
```

As we look above we see the command print *0x804909e. We see that it yields a value of 5 decimal. The binary at runtime puts the values inside the constants label to a respective memory address.

In this case we see that the pointer to 0x804909e or *0x804909e holds 5 decimal as we have stated above. An integer holds 4 bytes of data. The next value in our array will be stored in 0x80490a2. This memory location will hold the value of 8.

If we were to continue to advance through the array we would move 4 bytes to the next value and so forth. Remember each memory location in x86 32-bit assembly holds 4 bytes of data.

Let's hack!

```
noroot@noroot-gamma:~/Desktop/Code$ gdb -q indirect_addressing_with_registers
Reading symbols from indirect_addressing_with_registers...(no debugging symbols
found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/indirect_addressing_with_registers

Breakpoint 1, 0x08048074 in _start ()
(gdb) x/6d &constants
0x804909e:      5       8       17      44
0x80490ae:      50      52
(gdb) set *0x80490a2 = 66
(gdb) x/6d &constants
0x804909e:      5       66      17      44
0x80490ae:      50      52
```

After we broke on _start and ran, we examined the array like we did in our prior lesson. Here we hack the value at 0x80490a2 to 66 decimal instead of 8 decimal and we can see that we successfully changed one element of the array.

This lesson is very important to understand how arrays are ultimately stored in memory and how to manipulate and hack them. If you have any questions, please leave them in the comments below.

I look forward to seeing you all next week when we dive into programming our sixth assembly program!

Part 37 - ASM Program 6 [CMOV Instructions]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our sixth program we will demonstrate how we can work with CMOV instructions.

Before we dive into some code lets talk about CMOV is. CMOV can prevent the processor from utilizing the JMP instructions and speeds up the respective binary.

There are unsigned CMOV instructions such as:

CMOVA or CMOVNBE = Above [Carry Flag or Zero Flag = 0]

CMOVAE or CMOVNB = Above Or Equal [Carry Flag = 0]

CMOVNC = Not Carry [Carry Flag = 0]

CMOVB or CMOVNAE = Below [Carry Flag = 1]

CMOVC = Carry [Carry Flag = 1]

CMOVBE or CMOVNA = Below Or Equal [Carry Flag or Zero Flag = 1]

CMOVE or CMOVZ = Equal [Zero Flag = 1]

CMOVNE or CMOVNZ = Not Equal [Zero Flag = 0]

CMOVP or CMOVPE = Parity [Parity Flag = 1]

CMOVNP or CMOVPO = Not Parity [Parity Flag =0]

There are also signed CMOV instructions such as:

CMOVGE or CMOVNL = Greater Or Equal [Sign Flag xor Overflow Flag = 0]

CMOVL or CMOVNGE = Less [Sign Flag xor Overflow Flag = 1]

CMOVLE or CMOVNG = Less Or Equal [Sign Flag xor Overflow Flag or ZF = 1]

CMOVO = Overflow [Overflow Flag = 1]

CMOVNO = Not Overflow [Overflow Flag = 0]

CMOVS = Sign NEGATIVE [Sign Flag = 1]

CMOVNS = Not Sign POSITIVE [Sign Flag = 0]

Keep in mind to review the relationships between the unsigned and signed operations. The unsigned instructions utilize the CF, ZF and PF to determine the difference between the two operands where the signed instructions utilize the SF and OF to indicate the condition of the comparison between the operands.

If you need a refresher on the flag please review Part 14 on Flags in this series.

The CMOV instructions rely on a mathematical instruction that sets the EFLAGS register to operate and therefore saves the programmer to use JMP statements after the compare statement. Lets examine some source code.

```
1  #cmov_instructions:  conditional move instruction
2
3
4  .section .data
5      result:
6          .asciz "The smallest value is "
7      lr:
8          .ascii ".\n"
9      constants:
10         .int 43, 144, 32, 549, 600, 7, 10, 11
11
12 .section .bss
13     .comm answer, 1
14
15 .section .text
16     .globl _start
17
18 _start:
19     nop                             #used for debugging purposes
20
21     movl constants, %ebx            #mov array values into ebx
22     movl $1, %edi                   #load 2nd index constants label
23
24 find_smallest_value:
25     movl constants(, %edi, 4), %eax #mov value 4 bytes from constants
26     cmp %ebx, %eax                  #compare ebx to eax
27     cmovb %eax, %ebx                #compare below eax to ebx
28     inc %edi                        #increment edi to move through array
29     cmp $8, %edi                    #check where we are in array
30     jne find_smallest_value         #jne to beginning of loop
31     addl $0x30, %ebx                #convert int to ascii
32     movl %ebx, answer               #move new value of ebx to answer label
33
34     movl $4, %eax                   #sys_write
35     movl $1, %ebx                   #stdout
36     movl $result, %ecx              #mov result into ecx
37     movl $23, %edx                  #mov 23 bytes into edx
38     int $0x80                       #call sys_write
39
40     movl $4, %eax                   #sys_write
41     movl $1, %ebx                   #stdout
42     movl $answer, %ecx              #mov answer label into ecx
43     movl $1, %edx                   #mov 1 byte into edx
44     int $0x80                       #call sys_write
45
46     movl $4, %eax                   #sys_write
47     movl $1, %ebx                   #stdout
48     movl $lr, %ecx                  #mov lr label into ecx
49     movl $2, %edx                   #mov 1 byte into edx
50     int $0x80                       #call sys_write
51
52 exit:
53     movl $1, %eax                   #sys_exit system call
54     movl $0, %ebx                   #exit code 0 successful execution
55     int $0x80                       #call sys_exit
```

Ok lets begin with lines 21 and 22. This is nothing new that we have experienced as we are simply moving the array into ebx.

On line 24 we see the find_smallest_value function to where we are cycling through the array and using the CMOVB to find the lowest value ultimately.

We see **cmp %ebx, %eax** to which cmp subtracts the first operand from the second and sets the EFLAGS register appropriately. At this point the cmovb is used to replace the value in ebx with the value in eax if the value is smaller than what was originally in the ebx register.

After we exit the loop we see three sets of sys_writes to first display our message, second to display our converted integer to ascii value and then finally a period and line feed.

Keep in mind to assemble we type:

as –32 -o cmov_instructions.o cmov_instructions.s

To link the object file we type:

ld -m elf_i386 -o cmov_instructions cmov_instructions.o

I look forward to seeing you all next week when we dive into debugging our sixth assembly program!

Part 38 - ASM Debugging 6 [CMOV Instructions]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Lets re-examine some source code.

```
#cmov_instructions:  conditional move instruction


.section .data
    result:
        .asciz "The smallest value is "
    lr:
        .ascii ".\n"
    constants:
        .int 43, 144, 32, 549, 600, 7, 10, 11

.section .bss
    .comm answer, 1

.section .text
    .globl _start

_start:
    nop                             #used for debugging purposes

    movl constants, %ebx            #mov array values into ebx
    movl $1, %edi                   #load 2nd index constants label

find_smallest_value:
    movl constants(, %edi, 4), %eax #mov value 4 bytes from constants
    cmp %ebx, %eax                  #compare ebx to eax
    cmovb %eax, %ebx                #compare below eax to ebx
    inc %edi                        #increment edi to move through array
    cmp $8, %edi                    #check where we are in array
    jne find_smallest_value         #jne to beginning of loop
    addl $0x30, %ebx                #convert int to ascii
    movl %ebx, answer               #move new value of ebx to answer label

    movl $4, %eax                   #sys_write
    movl $1, %ebx                   #stdout
    movl $result, %ecx              #mov result into ecx
    movl $23, %edx                  #mov 23 bytes into edx
    int $0x80                       #call sys_write

    movl $4, %eax                   #sys_write
    movl $1, %ebx                   #stdout
    movl $answer, %ecx              #mov answer label into ecx
    movl $1, %edx                   #mov 1 byte into edx
    int $0x80                       #call sys_write

    movl $4, %eax                   #sys_write
    movl $1, %ebx                   #stdout
    movl $lr, %ecx                  #mov lr label into ecx
    movl $2, %edx                   #mov 1 byte into edx
    int $0x80                       #call sys_write

exit:
    movl $1, %eax                   #sys_exit system call
    movl $0, %ebx                   #exit code 0 successful execution
    int $0x80                       #call sys_exit
```

Lets break on 0x08048092 which is line 31. Lets do a r to run and then type **print $ebx**. We can see the value of 7.

```
pc@pc-delta:~/Desktop/Code$ gdb -q ./cmov_instructions
Reading symbols from ./cmov_instructions...(no debugging symbols found)...done.
(gdb) b *0x08048092
Breakpoint 1 at 0x8048092
(gdb) r
Starting program: /home/pc/Desktop/Code/cmov_instructions

Breakpoint 1, 0x08048092 in find_smallest_value ()
(gdb) print $ebx
$1 = 7
```

Ok now lets break on 0x080480b1 which is line 46. Remember when we are examining the value of **answer**, it has been converted to its ascii printable equivalent so in order to see the value of '7' you would type **x/1c &answer**.

```
(gdb) b *0x080480b1
Breakpoint 2 at 0x80480b1
(gdb) s
Single stepping until exit from function find_smallest_value,
which has no line number information.
The smallest value is
Breakpoint 2, 0x080480b1 in find_smallest_value ()
(gdb) x/1c &answer
0x8049122 <answer>:     55 '7'
```

I look forward to seeing you all next week when we dive into hacking our sixth assembly program!

Part 39 - ASM Hacking 6 [CMOV Instructions]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's bring the binary into gdb.

```
Reading symbols from cmov_instructions...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/pc/Desktop/cmov_instructions

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:     nop
   0x08048075 <+1>:     mov    0x8049102,%ebx
   0x0804807b <+7>:     mov    $0x1,%edi
End of assembler dump.
(gdb) si
0x08048075 in _start ()
(gdb) si
0x0804807b in _start ()
(gdb) si
0x08048080 in find_smallest_value ()
```

Let's now run the binary. We see that the smallest value is 7 which is expected. Our final bit of instruction in this tutorial will teach you how to jump to any part of the execution that you so choose.

```
(gdb) s
Single stepping until exit from function find_smallest_value,
which has no line number information.
The smallest value is 7.
0x080480dd in exit ()
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pc/Desktop/cmov_instructions

Breakpoint 1, 0x08048074 in _start ()
(gdb) set $eip = 0x080480dd
(gdb) s
Single stepping until exit from function exit,
which has no line number information.
[Inferior 1 (process 22023) exited normally]
(gdb)
```

We **set $eip = 0x080480dd** which is the exit routine. We see now that it bypasses all of the code from the nop instruction when we broke on _start. You now can use this command to jump anywhere inside of any binary within the debugger.

I look forward to seeing you all next week when we wrap up our tutorial series.

Part 40 - Conclusion

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

This has been an extensive and hopefully beneficial tutorial series for you all. Understanding assembly language is so important to everyone when trying to understand how Malware works in addition to programming no matter bare-metal assembly, c, c++ or even Java, Python or iOS or Android development.

If you are looking to pursue a career in Reverse Engineering, assembly will be second nature to you. Most of us will pursue higher-level language development as computers and devices are significantly more powerful today which allows for rapid development languages.

I want to thank you all for joining me on this tutorial series and look forward to you all making an impact in the future of tomorrow!

The 32-bit ARM Architecture (Part 1)

Let's dive in rightaway!

Part 1 - The Meaning Of Life

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Why C++? I primarily develop in Python professionally as an Automator however with every day passing we see another Ransomware attack that further cripples society in a catastrophic way.

This course is a comprehensive series where we learn every facet of C++ and how it relates to the ARM 64 architecture as we will reverse engineer each step in ARM 64 assembly language to get a full understanding of the environment.

There are roughly over 2,000 hacks a day world-wide and so few who truly understand how the hacks are executed on a fundamental level. This course is going to take a very basic and step-by-step approach to understanding low-level architecture as it relates to the ARM 64.

In our next lesson we will set up our development environment.

Part 2 – Number Systems

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

At the core of the microprocessor are a series of binary numbers which are either +5V (on or 1) or 0V (off or 0). Each 0 or 1 represents a bit of information within the microprocessor. A combination of 8 bits results in a single byte.

Before we dive into binary, lets examine the familiar decimal. If we take the number 2017, we would understand this to be two thousand and seventeen.

Value	1000s	100s	10s	1s
Representation	10^3	10^2	10^1	10^0
Digit	2	0	1	7

Let's take a look at the binary system and the basics of how it operates.

Bit Number	b7	b6	b5	b4	b3	b2	b1	b0
Representation	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Weight	128	64	32	16	8	4	2	1

If we were to convert a binary number into decimal, we would very simply do the following. Lets take a binary number of 0101 1101 and as you can see it is 93 decimal.

Bit	Weight	Value
0	128	0
1	64	64
0	32	0
1	16	16
1	8	8
1	4	4
0	2	0
1	1	1

Adding the values in the value column gives us 0 + 64 + 0 + 16 + 8 + 4 + 0 + 1 = 93 decimal.

If we were to convert a decimal number into binary, we would check to see if a subtraction is possible relative to the highest order bit and if so, a 1 would be placed into the binary column to which the remainder would be carried into the next row. Let's consider the example of the decimal value of 120 which is 0111 1000 binary.

128	64	32	16	8	4	2	1
0	1	1	1	1	0	0	0

1)Can 128 fit inside of 120: No, therefore 0.

2)Can 64 fit inside of 120: Yes, therefore 1, then 120 – 64 = 56.

3)Can 32 fit inside of 56: Yes, therefore 1, then 56 – 32 = 24.

4)Can 16 fit inside of 24: Yes, therefore 1, then 24 – 16 = 8.

5)Can 8 fit inside of 8: Yes, therefore 1, then 8 – 8 = 0.

6)Can 4 fit inside of 0: No, therefore 0.

7)Can 2 fit inside of 0: No, therefore 0.

8)Can 1 fit inside of 0: No, therefore 0.

When we want to convert binary to hex we simply work with the following table.

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Lets convert a binary number such as 0101 1111 to hex. To do this we very simply look at the table and compare each nibble which is a combination of 4 bits. Keep in mind, 8 bits is equal to a byte and 2 nibbles are equal to a byte.

0101 = 5

1111 = F

Therefore 0101 1111 binary = 0x5f hex. The 0x notation denotes hex.

To go from hex to binary it's very simple as you have to simply do the opposite such as:

0x3a = 0011 1010

3 = 0011

A = 1010

It is important to understand that each hex digit is a nibble in length therefore two hex digits are a byte in length.

To convert from hex to decimal we do the following:

0x5f = 95

5 = 5 x 16^1 = 5 x 16 = 80

F = 15 x 16^0 = 15 x 1 = 15

Therefore we can see that 80 + 15 = 95 which is 0x5f hex.

Finally to convert from decimal to hex. Lets take the number 850 decimal which is 352 hex.

Division	Result (No Remainder)	Remainder	Remainder Multiplication
850 / 16	53	0.125	0.125 x 16 = 2
53 / 16	3	0.3125	0.3125 x 16 = 5
3 / 16	0	0.1875	0.1875 x 16 = 3

We put the numbers together from bottom to the top and we get 352 hex.

“Why the hell would I waste my time learning all this crap when the computer does all this for me!”

If you happen to know any reverse engineers please if you would take a moment and ask them the above question.

The reality is, if you do NOT have a very firm understanding of how all of the above works, you will have a hard time getting a grasp on how the ARM processor registers hold and manipulate data. You will also have a hard time getting a grasp on how the ARM processor deals with a binary overflow and it’s effect on how carry operations work nor will you understand how compare operations work or even the most basic operations of the most simple assembly code.

I am not suggesting you memorize the above, nor am I suggesting that you do a thousand examples of each. All I ask is that you take the time to really understand that literally everything and I mean everything goes down to binary bits in the processor.

Whether you are creating, debugging or hacking an Assembly, Python, Java, C, C++, R, JavaScript, or any other new language application that hits the street, ultimately everything MUST go down to binary 0 and 1 to which represent a +5V or 0V.

We as humans operate on the base 10 decimal system. The processor works on a base 16 (hex) system. The registers we are dealing with in conjunction with Linux are addressed in 32-bit sizes. When we begin discussion of the processor registers, we will learn that each are 32-bits wide (technically the BCM2837 are 64-bit wide however our version of Linux that we are working with is 32-bit therefore we only address 32-bits of each register).

Next week we will dive into binary addition! Stay tuned!

Part 3 – Binary Addition

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Binary addition can occur in one of four different fashions:

```
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0 (1) [One Plus One Equals Zero, Carry One]
```

Keep in mind the (1) means a carry bit. It very simply means an overflow.

Lets take the following 4-bit nibble example:

```
  0111
+ 0100
= 1011
```

We see an obvious carry in the 3rd bit. If the 8th bit had a carry then this would generate a carry flag within the CPU.

Let's examine an 8-bit number:

```
  01110000
+ 01010101
= 11000101
```

If we had:

```
     11110000
+    11010101
= (1)11000101
```

Here we see a carry bit which would trigger the carry flag within the CPU to be 1 or true. We will discuss the carry flag in later tutorials. Please just keep in mind this example to reference as it is very important to understand.

Next week we will dive into binary subtraction! Stay tuned!

Part 4 – Binary Subtraction

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Binary subtraction is nothing more than adding the negative value of the number to be subtracted. For example 8 + - 4, the starting point would be zero to which we move 8 points in the positive direction and then four points in the negative direction yielding a value of 4.

We represent a sign bit in binary to which bit 7 indicates the sign of number where 0 is positive and 1 is negative.

```
Sign Bit 7          Bits 0 – 6
1                   0 0 0 0 0 1 1
```

The above would represent -2.

We utilize the concept of twos compliment which inverts each bit and then finally adding 1.

Lets examine binary 2.

00000010

Invert the bits.

11111101

```
Add 1.
  11111101
+ 00000001
  11111110
```

Let's examine a subtraction operation:

```
   00000100   4 decimal
+  11111110   -2 decimal
(1)00000010   2 decimal
```

So what is the (1) you may ask, that is the overflow bit. In future tutorials we will examine what we refer to as the overflow flag and carry flag.

Next week we will dive into word lengths! Stay tuned!

Part 5 – Word Lengths

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The system on chip we are working with has a 32-bit ARM CPU. 32-bits is actually 4 bytes of information which make up a word.

If you remember my prior tutorial on x86 Assembly, a word was 16-bits. Every different architecture defines a word differently.

The most significant bit of a word for our ARM CPU is located at bit 31 therefore a carry is generated if an overflow occurs there.

The lowest address in our architecture starts at 0x00000000 and goes to 0xFFFFFFFF. The processor sees memory in word blocks therefore every 4 bytes. A memory address associated with the start of a word is referred to as a word boundary and is divisible by 4. For example here is our first word:

```
0x00000000
0x00000004
0x00000008
0x0000000C
```

So why is this important? There is the concept of fetching and executing to which the processor deals with instructions to which it must work in this fashion for proper execution.

Before we dive into coding assembly it is critical that you understand some basics of how the CPU operates. There will be a number of more lectures going over the framework so I appreciate everyone hanging in there!

Next week we will dive into registers! Stay tuned!

Part 6 – Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Our ARM microprocessor has internal storage which make any operation must faster as there is no external memory access needed. There are two modes, User and Thumb. We will be focusing on User Mode as we are ultimately focused on developing for a system on chip within a Linux OS rather than bare-metal programming which would be better suited on a microcontroller device.

In User Mode we have 16 registers and a CPSR register to which have a word length each which is 32-bits each or 8 bytes each.

Registers R0 to R12 are multi-purpose registers to which R13 – R15 have a unique purpose as well as the CPSR. Lets take a look at a simple table to illustrate.

```
R0 GPR (General-Purpose Register)
R1 GPR (General-Purpose Register)
R2 GPR (General-Purpose Register)
R3 GPR (General-Purpose Register)
R4 GPR (General-Purpose Register)
R5 GPR (General-Purpose Register)
R6 GPR (General-Purpose Register)
R7 GPR (General-Purpose Register)
R8 GPR (General-Purpose Register)
R9 GPR (General-Purpose Register)
R10 GPR (General-Purpose Register)
R11 GPR (General-Purpose Register)
R12 GPR (General-Purpose Register)
R13 Stack Pointer
R14 Link Register
R15 Program Counter
CPSR Current Program Status Register
```

It is critical that we understand registers in a very detailed way. At this point we understand R0 – R12 are general purpose and will be used to manipulate data as we build our programs and additionally when you are hacking apart or reverse engineering binaries from a hex dump on a cell phone or other ARM device, no matter what high-level language it is written in, it must ultimately come down to assembly which you need to understand registers and how they work to grasp and understand of any such aforementioned operation.

The chip we are working with is known as a load and store machine. This means we load a register with the contents of a register or memory location and we can store a register with the contents of a memory or register location. For example:

```
ldr, r4, [r10] @
    load r4 with the contents of r10, if r10 had the
decimal value of
    say 22, 22 would go to r4

str, r9, [r4] @
    store r9 contents into location in r4, if r9 had 0x02
hex,
    0x02 would be stored into location r4
```

The @ simply indicates to the compiler that what follows it on a given line is a comment and to be ignored.

The next few weeks we will take our time and look at each of the special purpose registers so you have a great understanding of what they do.

Next week we will dive into more information on the program counter! Stay tuned!

Part 7 – Program Counter

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

We will dive into the registers over the coming weeks to make sure you obtain a firm understand of their role and what they can do.

We begin with the PC or program counter. The program counter is responsible for directing the CPU to what instruction will be executed next. The PC literally holds the address of the instruction to be fetched next.

When coding you can refer to the PC as PC or R15 as register 15 is the program counter. You MUST treat it with care as you can set it wrong and crash the executable quite easily.

You can control the PC directly in code:

```
mov r15, 0x00000000
```

I would not suggest trying that as we are not in Thumb mode and that will cause a fault as you would be going to an OS area rather than designated program area.

Regarding our ARM processor, we follow the standard calling convention meaning params are passed by placing the param values into regs R0 – R3 before calling the subroutine and the subroutine returns a value by putting it in R0 before returning.

This is important to understand when we think about how execution flows when dealing with a stack operation and the link register which we will discuss in future tutorials.

When you are hacking or reversing a binary, controlling the PC is essential when you want to test for subroutine execution and learning about how the program flows in order to break it down and understand exactly what it is doing.

Next week we will dive into more information on the CPSR! Stay tuned!

Part 8 - CPSR

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The CPSR register stores information about the program and the results of a particular operation. Bits that are in the respective registers have pre-assigned conditions that are tested for an occurrence which are flags.

There are 32-bits that total this register. The highest 4 we are concerned with most which are:

Bit 31 – N = Negative Flag

Bit 30 – Z = Zero Flag

Bit 29 – C = Carry Flag (UNSIGNED OPERATIONS)

Bit 28 – V = Overflow flag (SIGNED OPERATIONS)

When the instruction completes the CPSR can get updated if it falls into one of the aforementioned scenarios. If one of the conditions occurs, a 1 goes into the respective bits.

There are two instructions that directly effect the CPSR flags which are CMP and CMN. CMP is compare such as:

```
CMP R1, R0 @ notational subtraction where R1 – R0 and if
the result is 0, bit 30 Z would be set to 1
```

The most logical command that usually follows is BEQ = branch if equal, meaning the zero flag was set and branches to another label within the code.

Regarding CMP, if two operands are equal then the result is zero. CMN makes the same comparison but with the second operand negated for example:

```
CMN R1, R0 @ R1 - (-R0) or R1 + R0
```

When dealing with the SUB command, the result would NOT update the CPSR you would have to use the SUBS command to make any flag update respectively.

Next week we will dive into more information on the Link Register! Stay tuned!

Part 9 - Link Register

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The Link Register, R14, is used to hold the return address of a function call.

When a BL (branch with link) instruction performs a subroutine call, the link register is set to the subroutine return address. BL jumps to another location in the code and when complete allows a return to the point right after the BL code section. When the subroutine returns, the link register returns the address back to the program counter.

The link register does not require the writes and reads of the memory containing the stack which can save a considerable percentage of execution time with repeated calls of small subroutines.

When BL has executed, the return address which is the address of the next instruction to be executed, is loaded into the LR or R14. When the subroutine has finished, the LR is copied directly to the PC (Program Counter) or R15 and code execution continues where it was prior in the sequential code source.

CODE TIME! Don't be discouraged if you don't understand everything in the code example here. It will become clear over the next few lessons.

```
1  @ LR Demo - Link Register Demo
2
3      .global _start
4
5  _start:
6      mov r7, #0x30       @ mov hex 30 into r7
7      b no_return         @ branch to no_return function which never returns
8
9  no_return:
10     mov r7, #1          @ mov decimal 1 into r7
11     bl my_function      @ go to my_function which after execution
12                         @ will after this
13                         @ bl loads the return address of wrap_up into lr
14 wrap_up:
15     mov r7, #0x12       @ mov hex 12 into r7
16     b exit              @ branch to exit
17
18 my_function:
19     mov r0, #10         @ mov decimal 10 into r10
20     mov pc, lr          @ mov the ret address into the program counter
21                         @ here is where we return back to where we first
22                         @ bl therefore we go to wrap_up
23
24 exit:
25     mov r7, #1          @ sys_exit
26     svc 0
```

To compile:

```
as -o lr_demo.o lr_demo.s

ld -o lr_demo lr_demo.o
```

The simple example I created here is pretty self-explanatory. We start and proceed to the **no_return** subroutine and proceed to the **my_function** subroutine then to the **wrap_up** subroutine and finally **exit**.

```
pi@pi-beta:~/Code $ gdb -q lr_demo
Reading symbols from lr_demo...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/Code/lr_demo

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r7, #48 ; 0x30
   0x00010058 <+4>:     b       0x1005c <no_return>
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r7, #48 ; 0x30
=> 0x00010058 <+4>:     b       0x1005c <no_return>
End of assembler dump.
(gdb) si
0x0001005c in no_return ()
(gdb) disas
Dump of assembler code for function no_return:
=> 0x0001005c <+0>:     mov     r7, #1
   0x00010060 <+4>:     bl      0x1006c <my_function>
End of assembler dump.
(gdb) si
0x00010060 in no_return ()
(gdb) disas
Dump of assembler code for function no_return:
   0x0001005c <+0>:     mov     r7, #1
=> 0x00010060 <+4>:     bl      0x1006c <my_function>
End of assembler dump.
(gdb) si
0x0001006c in my_function ()
(gdb) disas
Dump of assembler code for function my_function:
=> 0x0001006c <+0>:     mov     r0, #10
   0x00010070 <+4>:     mov     pc, lr
End of assembler dump.
(gdb) si
0x00010070 in my_function ()
(gdb) disas
Dump of assembler code for function my_function:
   0x0001006c <+0>:     mov     r0, #10
=> 0x00010070 <+4>:     mov     pc, lr
End of assembler dump.
```

It is necessary that we jump into GDB which is our debugger to see exactly what happens with each step:

As you can see with every step inside the debugger it shows you exactly the progression from **no_return** to **my_function** skipping **wrap_up** until the program counter gets the address from the link register.

```
(gdb) si
0x00010064 in wrap_up ()
(gdb) disas
Dump of assembler code for function wrap_up:
=> 0x00010064 <+0>:     mov     r7, #18
   0x00010068 <+4>:     b       0x10074 <exit>
End of assembler dump.
(gdb) si
0x00010068 in wrap_up ()
(gdb) disas
Dump of assembler code for function wrap_up:
   0x00010064 <+0>:     mov     r7, #18
=> 0x00010068 <+4>:     b       0x10074 <exit>
End of assembler dump.
(gdb) si
0x00010074 in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x00010074 <+0>:     mov     r7, #1
   0x00010078 <+4>:     svc     0x00000000
End of assembler dump.
(gdb) si
0x00010078 in exit ()
(gdb) disas
Dump of assembler code for function exit:
   0x00010074 <+0>:     mov     r7, #1
=> 0x00010078 <+4>:     svc     0x00000000
End of assembler dump.
(gdb) si
[Inferior 1 (process 20229) exited with code 012]
```

Here we see the progression from **wrap_up** to **exit**.

This is a fundamental operation when we see next week how the stack operates as the LR is an essential part of this process.

Next week we will dive into the Stack Pointer! Stay tuned!

Part 10 - Stack Pointer

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The Stack is an abstract data type to which is a LIFO (Last In First Out). When we push a value onto the stack it goes into the Stack Pointer and when it is popped off of the stack it pops the value off of the stack and into a register of your choosing.

CODE TIME! Again, don't be discouraged if you don't understand everything in the code example here. It will become clear over the next few lessons.

```
1  @ Stack Pointer Demo - Stack Pointer Demo
2
3      .global _start
4
5  _start:
6      mov r7, #0x30       @ mov hex 30 into r7
7      push {r7}           @ push 0x30 value onto the Stack
8      mov r7, #0x10       @ mov hex 10 into r7
9      pop {r7}            @ pop the value on the Stack, 0x30 back into r7
10
11 exit:
12     mov r7, #1          @ sys_exit
13     svc 0
14
```

To compile:

```
as -o sp_demo.o sp_demo.s

ld -o sp_demo sp_demo.o
```

Once again lets load the binary into GDB to see what is happening.

```
pi@pi-beta:~/Code $ gdb -q sp_demo
Reading symbols from sp_demo...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/Code/sp_demo

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r7, #48 ; 0x30
   0x00010058 <+4>:     push    {r7}            ; (str r7, [sp, #-4]!)
   0x0001005c <+8>:     mov     r7, #16
   0x00010060 <+12>:    pop     {r7}            ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3a0       0x7efff3a0
lr             0x0      0
pc             0x10054  0x10054 <_start>
cpsr           0x10     16
```

Lets step into one time.

```
(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r7, #48 ; 0x30
=> 0x00010058 <+4>:     push    {r7}            ; (str r7, [sp, #-4]!)
   0x0001005c <+8>:     mov     r7, #16
   0x00010060 <+12>:    pop     {r7}            ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x30     48
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3a0       0x7efff3a0
lr             0x0      0
pc             0x10058  0x10058 <_start+4>
cpsr           0x10     16
```

We see **hex 30** or **48 decimal** moved into **r7**. Lets step into again.

```
(gdb) si
0x0001005c in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r7, #48 ; 0x30
   0x00010058 <+4>:     push    {r7}            ; (str r7, [sp, #-4]!)
=> 0x0001005c <+8>:     mov     r7, #16
   0x00010060 <+12>:    pop     {r7}            ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x30     48
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff39c       0x7efff39c
lr             0x0      0
pc             0x1005c  0x1005c <_start+8>
cpsr           0x10     16
```

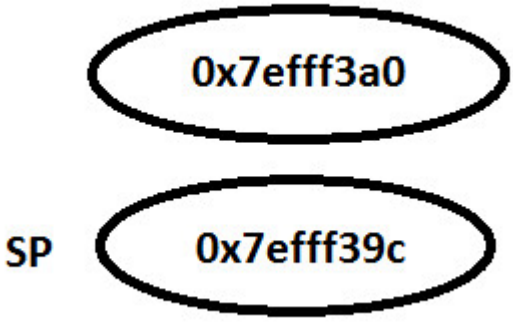
We see the value of the **sp** change from **0x7efff3a0** to **0xefff39c**. That is a movement backward **4 bytes**. Why the heck is the stack pointer going backward you may ask!

The answer revolves around the fact that the stack grows **DOWNWARD**. When we say the top of the stack you can imagine a series of plates being placed **BENEATH** of each other.

Originally the **sp** was at **0x7efff3a0**.



When we pushed **r7** onto the stack, the new value of the **Stack Pointer** is now **0x7efff39c** so we can see the Stack truly grows **DOWNWARD** in memory.



Now lets step into again.

```
(gdb) si
0x00010060 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r7, #48 ; 0x30
   0x00010058 <+4>:     push    {r7}            ; (str r7, [sp, #-4]!)
   0x0001005c <+8>:     mov     r7, #16
=> 0x00010060 <+12>:    pop     {r7}            ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x10     16
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff39c       0x7efff39c
lr             0x0      0
pc             0x10060  0x10060 <_start+12>
cpsr           0x10     16
```

We can see the value of **hex 10** or **decimal 16** moved into **r7**. Notice the **sp** did not change.

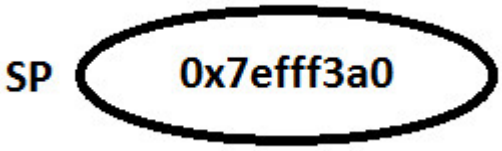
Before we step into again, lets look at the value inside the **sp**.

```
(gdb) x/x $sp
0x7efff39c:     0x30
```

Lets step into again.

```
(gdb) si
0x00010064 in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x00010064 <+0>:     mov     r7, #1
   0x00010068 <+4>:     svc     0x00000000
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x30     48
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3a0       0x7efff3a0
lr             0x0      0
pc             0x10064  0x10064 <exit>
cpsr           0x10     16
```

We see the value in the stack was popped off the stack and put back into **r7** therefore the value of **hex 30** is back in **r7** as well as the **sp** is back at **0x73fff3a0**.



Please take the time to type out the code, compile and link it and then step through the binary in GDB. Stack operations are critical to understanding Reverse Engineering and Malware Analysis as well as any debugging of any kind.

Next week we will dive into ARM Firmware Boot Procedures.

Part 11 - ARM Firmware Boot Procedures

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's take a moment to talk about what happens when we first power on our Raspberry Pi device.

As soon as the Pi receives power, the graphics processor is the first thing to run as the processor is held in a reset state to which the GPU starts executing code. The ROM reads from the SD card and reads **bootcode.bin** to which gets loaded into memory in C2 cache and turns on the rest of the RAM to which **start.elf** then loads.

The **start.elf** is an OS for the graphics processor and reads **config.txt** to which you can mod. The **kernel.img** then gets loaded into **0x8000** in memory which is the Linux kernel.

Once loaded, **kernel.img** turns on the CPU and starts running at **0x8000** in memory.

If we wanted, we could create our own **kernel.img** to which we can hard code machine code into a file and replace the original image and then reboot. Keep in mind the ARM word size is 32 bit long which go from bit 0 to 31.

As stated, when **kernel.img** is loaded the first byte, which is 8-bits, is loaded into address **0x8000**.

Lets open up a hex editor and write the following:

FE FF FF EA

Save the file as **kernel.img** and reboot.

"Ok nothing happens, this sucks!"

Actually something did happen, you created your first bare-metal firmware! Time to break out the champagne!

When the Pi boots, the below code when it reached **kernel.img** loads the following:

FE FF FF EA

@ address 0x8000, 0xfe gets loaded.

@ address 0x8001, 0xff gets loaded.

@ address 0x8002, 0xff gets loaded.

@ address 0x8003, 0xea gets loaded.

"So what the hell is really going on?"

This set of commands simply executes an infinite loop.

Review the datasheet:

https://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

The above code has 3 parts to it:

1)Conditional – Set To Always

2)Op Code – Branch

3)Offset – How Far To Move Within The Current Location

Condition – bits 31-28: 0xe or 1110

Op Code – bits 27-24: 0xa or 1010

Offset – bits 23-0 -2

I know this may be a lot to wrap your mind around however it is critical that you take the time and read the datasheet linked above. Do not cut corners if you truly have the passion to understand the above. READ THE DATASHEET!

I will go through painstaking efforts to break everything down step-by-step however there are exercises like the above that I am asking you to review the datasheet above so you learn how to better understand where to look when you are stuck on a particular routine or set of machine code. This is one of those times I ask you to please read and research the datasheet above!

“I’m bored! Why the hell does this crap matter?”

Glad you asked! The single most dangerous malware on planet earth today is that of the root-kit variety. If you do not have a basic understanding of the above, you will never begin to even understand what a root-kit is as you progress in your understanding.

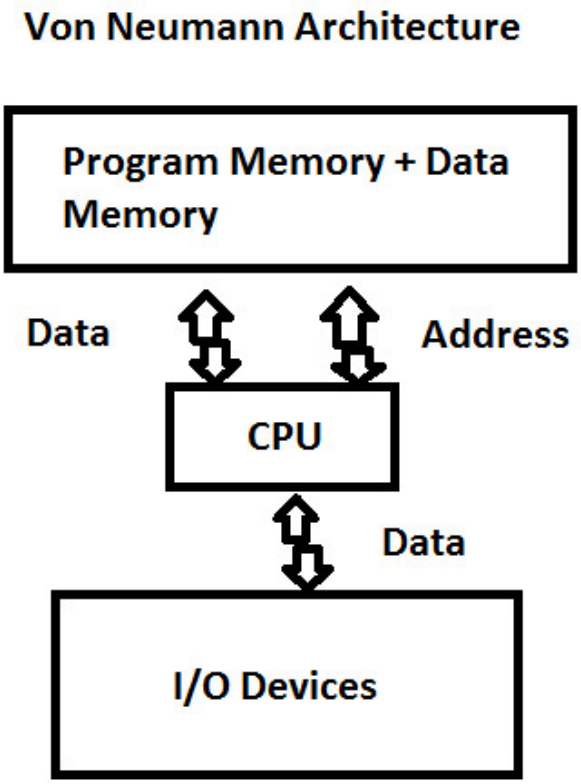
Anyone can simply replace the **kernel.img** file with their own hacked version and you can have total control over the entire process from boot.

Next week we will dive into the Von Neumann Architecture.

Part 12 - Von Neumann Architecture

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

ARM is a load and store machine to which the Arithmetic Logic Unit only operates on the registers themselves and any data that needs to be stored out to RAM, the control unit moves the data between memory and the registers which share the same data bus.



The CPU chip of this architecture holds a control unit and the arithmetic logic unit (along with some local memory) and the main memory is in the form of RAM sticks located on the motherboard.

A stored-program digital computer is one that keeps its program instructions, as well as its data, in read-write, random-access memory or RAM.

Next week we will dive into the Instruction Pipeline.

Part 13 - Instruction Pipeline

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The processor works with three separate phases which are:

1)**Fetch Phase** – The control unit grabs the instruction from memory and loads it into the instruction register.

2)**Decode Phase** – The control unit configures all of the hardware within the processor to perform the instruction.

3)**Execute Phase** – The processor computes the result of the instruction or operation.

When the processor processes instruction 1 we refer to it as being in the fetch phase. When the processor processes instruction 2, instruction 1 goes into the decode phase and instruction 2 goes into the fetch phase. When the processor processes instruction 3, instruction 2 goes into the decode stage and instruction 1 goes into the execute stage.



Keep in mind, if a branch instruction occurs, the pipeline might be flushed and start over again with a fresh set of cycles.

You now have a strong basis and background of ARM Assembly and how it works regarding its load and store capability between memory and the respective registers and the basics of how the instruction set flows.

Next week we will dive into our first C++ program!

Part 14 - ADD

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In ARM Assembly, we have three instructions that handle addition, the first being ADD, the second ADC (Add With Carry) and the final ADDS (Set Flag). This week we will focus on ADD.

Let's look at an example to illustrate:

```
#add - simple program to demonstrate the add command

        .global _start

_start:
        mov r1, #67             @ mov 67 decimal into r1
        mov r2, #53             @ mov 53 decimal into r2
        add r0, r1, r2          @ add r1 + r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

Here we see that we move decimal **67** into **r1** and decimal **53** into **r2**. We then **add r1** and **r2** and put the result into **r0**.

"So what the heck is all that and why should I care?"

This series is going to be unlike any other in it's class. The goal is to take small pieces of code and see exactly what it does. If you are going to understand how to reverse a binary or malware of any kind, it is critical that you understand the basics. Learning ARM Assembly basics will help you when reversing an iPhone or Android. This tutorial series is going to work to take extremely small bites of code and talk about:

1)**The Code**: (Here) we speak briefly about what the code does.

2)**The Debug**: We break down the binary in the GDB Debugger and step though each instruction and see what specifically it does to program flow, register values and flags.

3)**The Hack**: We hack a piece of the code to make it do whatever WE want!

This approach will allow you to spend just a few minutes each week to get a good grasp on what is going on behind the scenes.

Next week we will dive into Debugging ADD.

Part 15 - Debugging ADD

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our ADD example below:

```
#add - simple program to demonstrate the add command

        .global _start

_start:
        mov r1, #67             @ mov 67 decimal into r1
        mov r2, #53             @ mov 53 decimal into r2
        add r0, r1, r2          @ add r1 + r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

Again we see that we move decimal **67** into **r1** and decimal **53** into **r2**. We then **add r1** and **r2** and put the result into **r0**.

Let's compile:

as -o add.o add.s

ld -o add add.o

Let's bring into GDB to debug:

gdb -q add

```
Reading symbols from add...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) q
pi@pi-alpha:~/code $ gdb -q add
Reading symbols from add...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/add

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #67 ; 0x43
   0x00010058 <+4>:     mov     r2, #53 ; 0x35
   0x0001005c <+8>:     add     r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10054  0x10054 <_start>
cpsr           0x10     16
(gdb)
```

We can see that when we b **_start**, break on start and **r**, run we see the disassembly. If you do an **i r** we see the info registers where we notice our **cpsr** is **0x10**.

As we step again and info registers:

```
(gdb) si
0x00010058 in _start ()
(gdb) i r
r0             0x0      0
r1             0x43     67
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10058  0x10058 <_start+4>
cpsr           0x10     16
(gdb)
```

We notice **0x43** hex or **67** decimal into **r1**. We also notice that the flags are unchanged (**cpsr 0x10**).

Let's step again and info registers:

```
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r1, #67 ; 0x43
   0x00010058 <+4>:     mov     r2, #53 ; 0x35
=> 0x0001005c <+8>:     add     r0, r1, r2
End of assembler dump.
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0             0x78     120
r1             0x43     67
r2             0x35     53
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10060  0x10060 <exit>
cpsr           0x10     16
(gdb)
```

We can see **r0** now holds **0x78** hex or **120** decimal. We successfully saw the add instruction in place and we again notice that the flags register (**cpsr**) remains unchanged by this operation.

Next week we will dive into Hacking ADD.

Part 16 - Hacking ADD

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's again review our ADD example below:

```
#add - simple program to demonstrate the add command

        .global _start

_start:
        mov r1, #67             @ mov 67 decimal into r1
        mov r2, #53             @ mov 53 decimal into r2
        add r0, r1, r2          @ add r1 + r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

Let's debug:

```
pi@pi-alpha:~/code $ gdb -q add
Reading symbols from add...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/add

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #67 ; 0x43
   0x00010058 <+4>:     mov     r2, #53 ; 0x35
   0x0001005c <+8>:     add     r0, r1, r2
End of assembler dump.
```

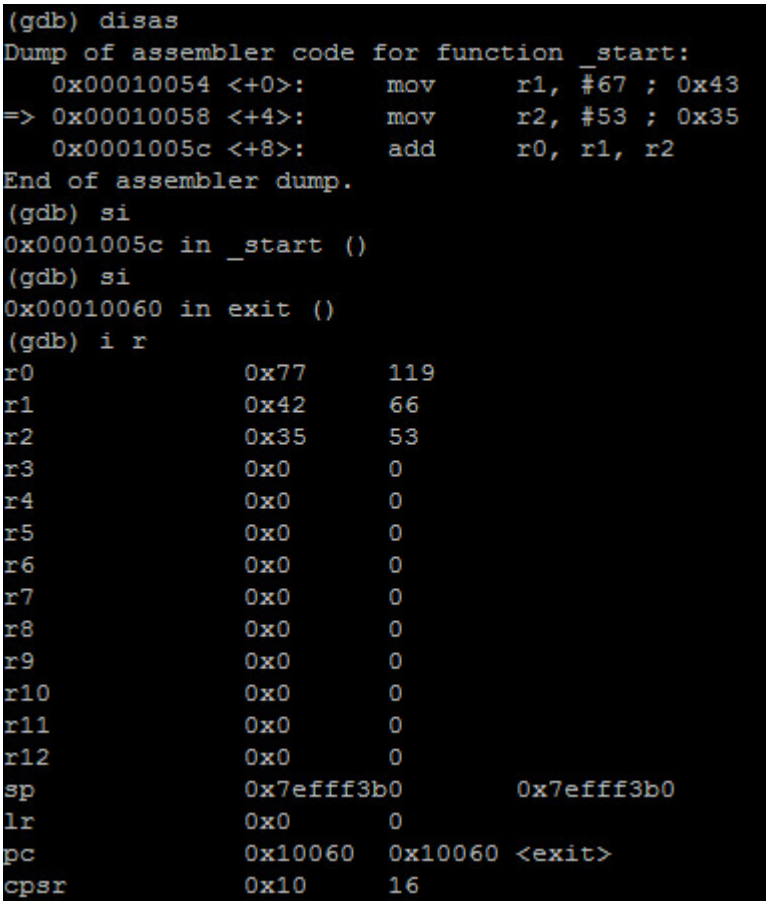
We see the value of **67** decimal is being moved into **r1** below:

```
(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r1, #67 ; 0x43
=> 0x00010058 <+4>:     mov     r2, #53 ; 0x35
   0x0001005c <+8>:     add     r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x43     67
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10058  0x10058 <_start+4>
cpsr           0x10     16
```

Let's hack! Lets set **r1 = 66**!

```
(gdb) set $r1 = 66
(gdb) i r
r0             0x0      0
r1             0x42     66
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10058  0x10058 <_start+4>
cpsr           0x10     16
```

Now we see we have hacked the program so when it adds the values it will have a different output. If you remember back to the last lecture, **r0 = 120**. Here we see we have hacked r1 and now the value of **r0** is **119**!



```
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r1, #67 ; 0x43
=> 0x00010058 <+4>:     mov     r2, #53 ; 0x35
   0x0001005c <+8>:     add     r0, r1, r2
End of assembler dump.
(gdb) si
0x0001005c in _start ()
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0             0x77     119
r1             0x42     66
r2             0x35     53
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10060  0x10060 <exit>
cpsr           0x10     16
```

This is the power of understanding assembly. This is a VERY simple example however with each new series as I have stated we will create a program, debug and hack it.

This combination of instructions will help you to get hands on experience when learning how to have absolute control over an application and in the case of malware reverse engineering gives you the ability to make the binary do exactly what you want!

Next week we will dive into ADDS.

Part 17 - ADDS

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

ADDS is the same as ADD except it sets the flags accordingly in the CPSR.

Let's look at an example to illustrate:

```
#adds - simple program to demonstrate the adds command

        .global _start

_start:
        mov r1, #100            @ mov 100 decimal into r1
        mov r2, #4294967295     @ mov 4294957295 decimal into r2

        adds r0, r1, r2         @ adds r1 + r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**. We then **add r1** and **r2** and place in **r0.**

We see **adds** which sets the flags in the CPSR. We have to remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start to debug and hack this example in the coming tutorials.

You can compile the above by:

```
as -o adc.o adc.s
ld -o adc adc.o
```

We need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

Therefore if the value in binary was **0110** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag SET

Carry Flag SET

Overflow Flag NOT Set

It is critical that you compile, debug and hack each exercise in order to understand what is going on here.

Next week we will dive into Debugging ADDS.

Part 18 – Debugging ADDS

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code:

```
#adds - simple program to demonstrate the adds command

        .global _start

_start:
        mov r1, #100            @ mov 100 decimal into r1
        mov r2, #4294967295     @ mov 4294957295 decimal into r2

        adds r0, r1, r2         @ adds r1 + r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We again **add 100** decimal into **r1**, **4,294,967,295** into **r2**. We then **add r1** and **r2** and place in **r0.**

Lets debug:

```
pi@pi-alpha:~/code $ gdb -q adds
Reading symbols from adds...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adds

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #100        ; 0x64
   0x00010058 <+4>:     mvn     r2, #0
   0x0001005c <+8>:     adds    r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10054  0x10054 <_start>
cpsr           0x10     16
(gdb)
```

We again see **adds** which sets the flags in the CPSR. We have to remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start

to debug and hack this example in the coming tutorials.

We need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **10 hex**. **10 hex** in binary is **00010000**.

Therefore if the value in binary was **00010000** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag NOT SET

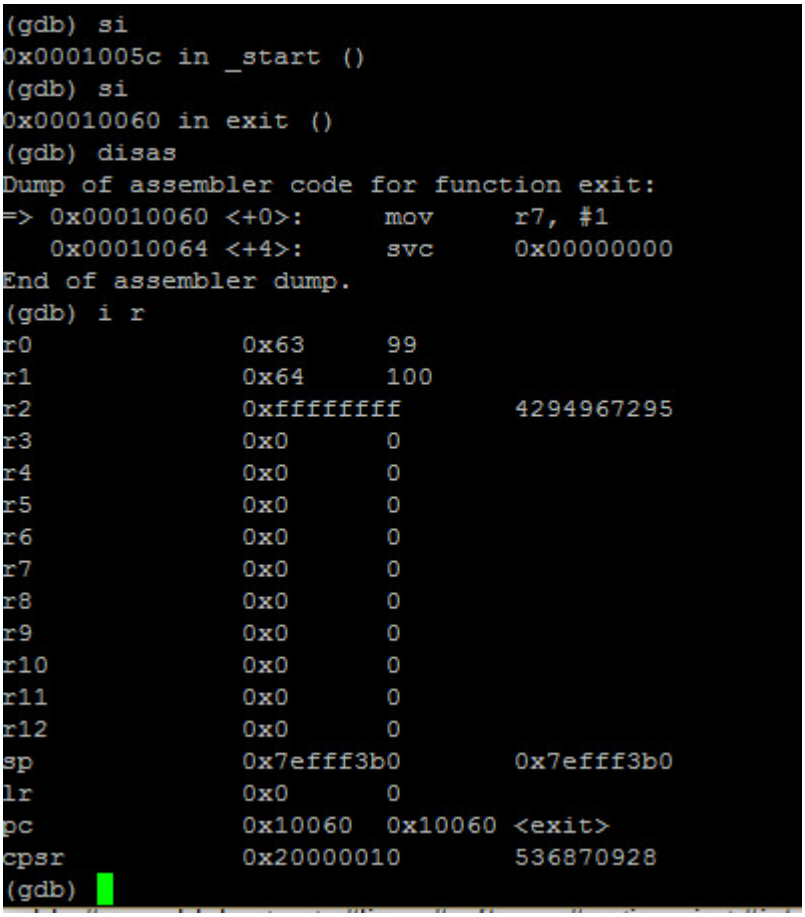
Overflow Flag Set

There is nothing in code above which set the **Overflow Flag** however in it's natural state upon executing this binary it is set.

Lets step through the program:

```
(gdb) si
0x00010058 in _start ()
(gdb) i r
r0             0x0        0
r1             0x64       100
r2             0x0        0
r3             0x0        0
r4             0x0        0
r5             0x0        0
r6             0x0        0
r7             0x0        0
r8             0x0        0
r9             0x0        0
r10            0x0        0
r11            0x0        0
r12            0x0        0
sp             0x7efff3b0         0x7efff3b0
lr             0x0        0
pc             0x10058    0x10058 <_start+4>
cpsr           0x10       16
(gdb)
```

We see **64 hex** or **100 decimal** moved into **r1** as expected. No change in the **CPSR**. Lets step some more.



We see the addition that transpires above and notice the value in **r0** is **99 decimal** after **100 decimal** and **4294967295 decimal** were added together. How is that possible? The answer is simple, we overflowed the 32-bit register of **r0** from this addition.

If we examine the **CPSR** we now see **20000010 hex** or **0010 0000 0000 0000 0000 0000 0001 0000 binary**. We only have to focus on the most significant bits which are **0010**:

The value in binary is **0010** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag SET

Overflow Flag NOT Set

We see that the **Carry Flag** was set and the **Overflow Flag** was NOT set. Why is that?

The **Carry Flag** is a flag set when two **unsigned numbers** were added and the result is larger than the register where it is saved. We are dealing with a 32-bit register. We are also dealing with unsigned numbers therefore the **CF** is set and the **OF** was not as the **OF** flag deals with **signed numbers**.

Next week we will dive into Hacking ADDS.

Part 19 – Hacking ADDS

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's once again re-examine our code:

```
#adds - simple program to demonstrate the adds command

        .global _start

_start:
        mov r1, #100            @ mov 100 decimal into r1
        mov r2, #4294967295     @ mov 4294957295 decimal into r2

        adds r0, r1, r2         @ adds r1 + r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We again **add 100** decimal into **r1**, **4,294,967,295** into **r2**. We then **add r1** and **r2** and place in **r0.**

Lets debug:

```
pi@pi-alpha:~/code $ gdb -q adds
Reading symbols from adds...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adds

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #100        ; 0x64
   0x00010058 <+4>:     mvn     r2, #0
   0x0001005c <+8>:     adds    r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10054  0x10054 <_start>
cpsr           0x10     16
(gdb)
```

We again see **adds** which sets the flags in the CPSR. We have to remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start

to debug and hack this example in the coming tutorials.

We need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **10 hex**. **10 hex** in binary is **0001**.

Therefore if the value in binary was **0001** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag NOT SET

Overflow Flag Set

Lets take a look if we step again:

```
pi@pi-alpha:~/code $ gdb -q adds
Reading symbols from adds...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adds

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #100        ; 0x64
   0x00010058 <+4>:     mvn     r2, #0
   0x0001005c <+8>:     adds    r0, r1, r2
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r1, #100        ; 0x64
   0x00010058 <+4>:     mvn     r2, #0
=> 0x0001005c <+8>:     adds    r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x64     100
r2             0xffffffff       4294967295
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x1005c  0x1005c <_start+8>
cpsr           0x10     16
```

We see **4294967295 decimal** or **0xffffffff** in **r2**. We know if we step again we will cause the CPSR to change from 0001 to 0010 which means:

The value in binary is **0010** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag SET

Overflow Flag NOT Set

This action sets the carry flag. However lets hack:

```
(gdb) set $r2 = 1
(gdb) i r
r0             0x0        0
r1             0x64       100
r2             0x1        1
r3             0x0        0
r4             0x0        0
r5             0x0        0
r6             0x0        0
r7             0x0        0
r8             0x0        0
r9             0x0        0
r10            0x0        0
r11            0x0        0
r12            0x0        0
sp             0x7efff3b0         0x7efff3b0
lr             0x0        0
pc             0x1005c    0x1005c <_start+8>
cpsr           0x10       16
```

We hacked **r2** and changed the value to **1 decimal** and **0x1 hex**. NOW we know before the **CPSR** went to **0010** last time however now that we hacked this, lets see what happens to the **CPSR** when we step.

```
(gdb) i r
r0             0x65       101
r1             0x64       100
r2             0x1        1
r3             0x0        0
r4             0x0        0
r5             0x0        0
r6             0x0        0
r7             0x0        0
r8             0x0        0
r9             0x0        0
r10            0x0        0
r11            0x0        0
r12            0x0        0
sp             0x7efff3b0         0x7efff3b0
lr             0x0        0
pc             0x10060    0x10060 <exit>
cpsr           0x10       16
```

BAM! We hacked it and see **r0** is **101** and therefore did NOT trigger the carry flag and kept the **CPSR** at **0x10 hex** which means **0001 binary** which means:

Therefore if the value in binary was **0001** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag NOT SET

Overflow Flag Set

It is so important that you understand this lesson in its entirety. If not, please review the last two weeks lessons.

Next week we will dive into ADC.

Part 20 – ADC

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

ADC is the same as ADD except it adds a 1 if the carry flag is set. We need to pay particular attention to the CPSR or Status Register when we work with ADC.

Let's look at an example to illustrate:

```
#adc - simple program to demonstrate the adc command

        .global _start

_start:
        mov r1, #100            @ mov 100 decimal into r1
        mov r2, #4294967295     @ mov 4294957295 decimal into r2
        mov r3, #100            @ mov 100 decimal into r3
        mov r4, #100            @ mov 100 decimal into r4

        adds r0, r1, r2         @ adds r1 + r2 and store in r0
        adc r5, r3, r4          @ adc r3 + r4 and store in r5

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**, **100** decimal into **r3** and **100** decimal into **r4**. We then **add r1** and **r2** and place in **r0** and then **add r3** and **r4** and place into **r5**.

We see **adds** which sets the flags in the CPSR. We have to once again remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start to debug and hack this example in the coming tutorials.

You can compile the above by:

```
as -o adc.o adc.s
ld -o adc adc.o
```

I want you to ask yourself what is going to happen when **r3(100 decimal)** is added to **r4(100 decimal)**? What do you think the value of **r5** will be with the above example of setting the flags with the adds result? Think about the first sentence in this tutorial and keep this in mind for the next tutorial.

Next week we will dive into Debugging ADC.

Part 21 – Debugging ADC

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

To recap, ADC is the same as ADD except it adds a 1 if the carry flag is set. We need to pay particular attention to the CPSR or Status Register when we work with ADC.

Let's review our code:

```
#adc - simple program to demonstrate the adc command

        .global _start

_start:
        mov r1, #100            @ mov 100 decimal into r1
        mov r2, #4294967295     @ mov 4294957295 decimal into r2
        mov r3, #100            @ mov 100 decimal into r3
        mov r4, #100            @ mov 100 decimal into r4

        adds r0, r1, r2         @ adds r1 + r2 and store in r0
        adc r5, r3, r4          @ adc r3 + r4 and store in r5

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**, **100** decimal into **r3** and **100** decimal into **r4**. We then **add r1** and **r2** and place in **r0** and then **add r3** and **r4** and place into **r5**.

We see **adds** which sets the flags in the CPSR. We have to once again remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start to debug and hack this example in the coming tutorials.

Last week I raised a question where I wanted you to ask yourself what is going to happen when **r3(100 decimal)** is added to **r4(100 decimal)**? What do you think the value of **r5** will be with the above example of setting the flags with the adds result?

```
pi@pi-alpha:~/code $ gdb -q adc
Reading symbols from adc...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adc

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #100        ; 0x64
   0x00010058 <+4>:     mvn     r2, #0
   0x0001005c <+8>:     mov     r3, #100        ; 0x64
   0x00010060 <+12>:    mov     r4, #100        ; 0x64
   0x00010064 <+16>:    adds    r0, r1, r2
   0x00010068 <+20>:    adc     r5, r3, r4
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) si
0x00010060 in _start ()
(gdb) si
0x00010064 in _start ()
(gdb) si
0x00010068 in _start ()
(gdb) si
0x0001006c in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x0001006c <+0>:     mov     r7, #1
   0x00010070 <+4>:     svc     0x00000000
End of assembler dump.
(gdb) i r
r0             0x63     99
r1             0x64     100
r2             0xffffffff       4294967295
r3             0x64     100
r4             0x64     100
r5             0xc9     201
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x1006c  0x1006c <exit>
cpsr           0x20000010       536870928
(gdb)
```

Ok so we add **100 decimal** and **100 decimal** together in **r3** and **r4** and we get **201 decimal** in **r5**! Is something broken? ADC is the same as ADD except it adds a 1 if the carry flag is set. Therefore we get the extra 1 in **r5**.

We again need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **20000010 hex**. The most significant bits of **20000010 hex** in binary is **0010**.

Therefore if the value in binary was **0010** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT Set

Carry Flag SET

Overflow Flag NOT Set

As we can clearly see the carry flag was set. I hope you can digest and understand each of these very simple operations and how they have an effect on the CPSR.

Next week we will dive into Hacking ADC.

Part 22 – Hacking ADC

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

To recap again, ADC is the same as ADD except it adds a 1 if the carry flag is set. We need to pay particular attention to the CPSR or Status Register when we work with ADC.

Let's again review our code:

```
#adc - simple program to demonstrate the adc command

        .global _start

_start:
        mov r1, #100            @ mov 100 decimal into r1
        mov r2, #4294967295     @ mov 4294957295 decimal into r2
        mov r3, #100            @ mov 100 decimal into r3
        mov r4, #100            @ mov 100 decimal into r4

        adds r0, r1, r2         @ adds r1 + r2 and store in r0
        adc r5, r3, r4          @ adc r3 + r4 and store in r5

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**, **100** decimal into **r3** and **100** decimal into **r4**. We then **add r1** and **r2** and place in **r0** and then **add r3** and **r4** and place into **r5**.

```
pi@pi-alpha:~/code $ gdb -q adc
Reading symbols from adc...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adc

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #100        ; 0x64
   0x00010058 <+4>:     mvn     r2, #0
   0x0001005c <+8>:     mov     r3, #100        ; 0x64
   0x00010060 <+12>:    mov     r4, #100        ; 0x64
   0x00010064 <+16>:    adds    r0, r1, r2
   0x00010068 <+20>:    adc     r5, r3, r4
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r1, #100        ; 0x64
=> 0x00010058 <+4>:     mvn     r2, #0
   0x0001005c <+8>:     mov     r3, #100        ; 0x64
   0x00010060 <+12>:    mov     r4, #100        ; 0x64
   0x00010064 <+16>:    adds    r0, r1, r2
   0x00010068 <+20>:    adc     r5, r3, r4
End of assembler dump.
(gdb) si
```

We run the program and step to where we move **4,294,967,295** into **r2**. Let's hack that value in **r2** and change it to **100 decimal**.

```
0x0001005c in _start ()
(gdb) set $r2 = 100
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:     mov     r1, #100        ; 0x64
   0x00010058 <+4>:     mvn     r2, #0
=> 0x0001005c <+8>:     mov     r3, #100        ; 0x64
   0x00010060 <+12>:    mov     r4, #100        ; 0x64
   0x00010064 <+16>:    adds    r0, r1, r2
   0x00010068 <+20>:    adc     r5, r3, r4
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x64     100
r2             0x64     100
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x1005c  0x1005c <_start+8>
cpsr           0x10     16
```

Let's step a few more times:

```
(gdb) si
0x00010060 in _start ()
(gdb) si
0x00010064 in _start ()
(gdb) si
0x00010068 in _start ()
(gdb) si
0x0001006c in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x0001006c <+0>:     mov     r7, #1
   0x00010070 <+4>:     svc     0x00000000
End of assembler dump.
(gdb) i r
r0             0xc8     200
r1             0x64     100
r2             0x64     100
r3             0x64     100
r4             0x64     100
r5             0xc8     200
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x1006c  0x1006c <exit>
cpsr           0x10     16
```

Ok so now we add **100 decimal** and **100 decimal** together in **r3** and **r4** and we get **200 decimal** in **r5**! Do you remember last week when we had **201**? Let's examine the CPSR below.

We again need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **10 hex**. The most significant bits of **10 hex** in binary is **0001**.

Therefore if the value in binary was **0001** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT Set

Carry Flag NOT SET

Overflow Flag Set

As we can clearly see the carry flag was NOT set. I hope you can digest and understand each of these very simple operations and how they have an effect on the CPSR. Please take the time and review last weeks lesson for comparison.

Next week we will dive into SUB.

Part 23 – SUB

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Subtraction in ARM has four instructions which are SUB, SBC, RSB and RSC. We will start today with SUB.

Please keep in mind when you add the S suffix on the end of each such as SUBS, SBCS, RSBS, RSCS, it will affect the flags. We have spent enough time on flags in the prior lessons so that you should have a firm grasp on this now.

Let's examine an example of SUB:

```
#sub - simple program to demonstrate the sub command

        .global _start

_start:
        mov r1, #67             @ mov 67 decimal into r1
        mov r2, #53             @ mov 53 decimal into r2
        sub r0, r1, r2          @ sub r1 - r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

To compile:

```
as -o sub.o sub.s
ld -o sub sub.o
```

We simply take **67 decimal** and move into **r1** and **53 decimal** and move into **r2** and subtract r1 – r2 and put the result in **r0**.

Next week we will dive into SUB debugging.

Part 24 – Debugging SUB

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

As stated, subtraction in ARM has four instructions which are SUB, SBC, RSB and RSC. We will start today with SUB.

Please keep in mind when you add the S suffix on the end of each such as SUBS, SBCS, RSBS, RSCS, it will affect the flags. We have spent enough time on flags in the prior lessons so that you should have a firm grasp on this now.

Let's re-examine our example of SUB:

```
#sub - simple program to demonstrate the sub command

        .global _start

_start:
        mov r1, #67             @ mov 67 decimal into r1
        mov r2, #53             @ mov 53 decimal into r2
        sub r0, r1, r2          @ sub r1 - r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We simply take **67 decimal** and move into **r1** and **53 decimal** and move into **r2** and subtract r1 – r2 and put the result in **r0**.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q sub
Reading symbols from sub...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/sub

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #67 ; 0x43
   0x00010058 <+4>:     mov     r2, #53 ; 0x35
   0x0001005c <+8>:     sub     r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10054  0x10054 <_start>
cpsr           0x10     16
```

As we can see the registers are clear. Lets step through and see what the value of **r0** becomes.

```
(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0             0xe      14
r1             0x43     67
r2             0x35     53
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10060  0x10060 <exit>
cpsr           0x10     16
```

As you can see above **r0** now has **decimal 14** which works as expected.

Next week we will dive into SUB hacking.

Part 25 – Hacking SUB

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

As stated, subtraction in ARM has four instructions which are SUB, SBC, RSB and RSC. We will start today with SUB.

Please keep in mind when you add the S suffix on the end of each such as SUBS, SBCS, RSBS, RSCS, it will affect the flags. We have spent enough time on flags in the prior lessons so that you should have a firm grasp on this now.

Let's re-examine our example of SUB:

```
#sub - simple program to demonstrate the sub command

        .global _start

_start:
        mov r1, #67             @ mov 67 decimal into r1
        mov r2, #53             @ mov 53 decimal into r2
        sub r0, r1, r2          @ sub r1 - r2 and store in r0

exit:
        mov r7, #1              @ exit syscall
        swi 0
```

We simply take **67 decimal** and move into **r1** and **53 decimal** and move into **r2** and subtract r1 – r2 and put the result in **r0**.

Let's hack.

```
pi@pi-alpha:~/code $ gdb -q sub
Reading symbols from sub...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/sub

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #67 ; 0x43
   0x00010058 <+4>:     mov     r2, #53 ; 0x35
   0x0001005c <+8>:     sub     r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10054  0x10054 <_start>
cpsr           0x10     16
```

As we can see the registers are clear. Lets step through and see what the value of **r0** becomes when we do a little hacking.

```
pi@pi-alpha:~/code $ gdb sub -q
Reading symbols from sub...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/sub

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:     mov     r1, #67 ; 0x43
   0x00010058 <+4>:     mov     r2, #53 ; 0x35
   0x0001005c <+8>:     sub     r0, r1, r2
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) set $r2 = 50
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0             0x11     17
r1             0x43     67
r2             0x32     50
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0       0x7efff3b0
lr             0x0      0
pc             0x10060  0x10060 <exit>
cpsr           0x10     16
(gdb)
```

As you can see above **r0** now has **decimal 17** which works as expected as we hacked the value of **r2** to **decimal 50** instead of **decimal 53**.

I want to thank you all for taking this journey to learn ARM Assembly. This is the end of the series as I encourage you all to take what you have learned and continue to work through the ARM instruction set and continue your progress.

This tutorial's purpose was to provide you a solid foundation in ARM Assembly and I believe we have done that. Thank you all and I look forward to seeing you all become future Reverse Engineers!

The 32-bit ARM Architecture (Part 2)

Let's dive in rightaway!

Part 1 – The Meaning Of Life Part 2

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Welcome to the ARM Reverse Engineering tutorial. This is the third tutorial series that I have done focusing on Assembly Language and Reverse Engineering.

The first series was on x86 Assembly and the second was on ARM Assembly. This series will be an expansion series on ARM focusing on ARM Reverse Engineering so rather than create programs directly in Assembly alone and then Reverse Engineer the binary in Assembly we will work with Assembly and C together and Reverse Engineer in Assembly so that you will get a flavor for a real-world series of applications and what it looks like disassembled.

We will not be working with GUI tools such as IDA Pro as we will be working with GDB in CLI shell. We will not be working in a traditional lab environment where we are going to put a binary into a debugger rather we are going to SSH into the ARM device and actually attach to a running process (PID) and Reverse Engineer the process as it is running.

The first 13 weeks will be an exact review of the ARM Assembly series as it is critical that we re-examine these concepts so that we have a very firm grasp when it comes time to reverse our binaries.

I wanted to bring back the original quote below before we get started...

“So if I go to college and learn Java will I make a million dollars and have nice things?”

I felt it necessary to start out this tutorial series with such a statement. This is NOT an attack on Java as I have used Java in Android Development, Spring and JavaEE. In today’s Agile environment, rapid-development is reality. With the increased challenges in both the commercial market and the government sector, software development will continue to focus on more robust libraries that will do more with less. React, Python, Java, C# and the like will continue to grow not shrink as the race for project completion augments with each passing second of time.

Like it or not, hardware is getting smaller and smaller and the trend is going from CISC to RISC. A CISC is your typical x86/x64 computer with a complex series of instructions. CISC computers will always exist however with the trend going toward cloud computing and the fact that RISC machines with a reduced instruction set are so enormously powerful today, they are the obvious choice for consumption.

How many cell phones do you think exist on earth today? Most of them are RISC machines. How many of you have a Smart TV or Amazon Echo or any number of devices considered part of the IOT or Internet Of Things? Each of these devices have one thing in common – they are RISC and all are primarily ARM based.

ARM is an advanced RISC machine. Compared to the very complex architecture of a CISC, most ARM systems today are what is referred to as a SoC or system on chip which is an integrated circuit which has all of the components of a computer and electronic system on a single chip. This includes RF functionality as well. These low-power embedded devices can run versions of Windows, Linux and many other advanced operating systems.

“Well who cares about ARM, you can call it anything you want, I know Java and that’s all I need to know cause when I program it works everywhere so I don’t have to worry about anything under the hood.”

I again just want you to reflect on the above statement for a brief moment. As every day continues to pass, more and more systems are becoming vulnerable to attack and compromise. Taking the time to understand what is going on under the hood can only help to curb this unfortunate reality.

This series will focus on ARM Reverse Engineering. We will work with a Raspberry Pi 3 which contains the Broadcom BCM2837 SoC with a 4x ARM Cortex-A53, 1.2GHz CPU and 1 GB LPDDR2 RAM. We will work with the Raspbian Jessie, Linux-based operating system. If you don’t own a Raspberry Pi 3, they are usually available for $35 on Amazon or any number of retailers. If you would like to learn more visit https://www.raspberrypi.org.

We will work solely in the terminal so no pretty pictures and graphics as we are keeping it to the hardcore bare-bones utilizing the GNU toolkit to compile and debug our code base.

Next week we will dive into the binary number system and compare and contrast it with decimal and hexadecimal so we have a proper framework of understanding to move forward.

Part 11 - Firmware Boot Procedures

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's take a moment to talk about what happens when we first power on our Raspberry Pi device.

As soon as the Pi receives power, the graphics processor is the first thing to run as the processor is held in a reset state to which the GPU starts executing code. The ROM reads from the SD card and reads **bootcode.bin** to which gets loaded into memory in C2 cache and turns on the rest of the RAM to which **start.elf** then loads.

The **start.elf** is an OS for the graphics processor and reads **config.txt** to which you can mod. The **kernel.img** then gets loaded into **0x8000** in memory which is the Linux kernel.

Once loaded, **kernel.img** turns on the CPU and starts running at **0x8000** in memory.

If we wanted, we could create our own **kernel.img** to which we can hard code machine code into a file and replace the original image and then reboot. Keep in mind the ARM word size is 32 bit long which go from bit 0 to 31.

As stated, when **kernel.img** is loaded the first byte, which is 8-bits, is loaded into address **0x800**.

Lets open up a hex editor and write the following:

FE FF FF EA

Save the file as **kernel.img** and reboot.

"Ok nothing happens, this sucks!"

Actually something did happen, you created your first bare-metal firmware! Time to break out the champagne!

When the Pi boots, the below code when it reached **kernel.img** loads the following:

FE FF FF EA

@ address 0x8000, 0xfe gets loaded.

@ address 0x8001, 0xff gets loaded.

@ address 0x8002, 0xff gets loaded.

@ address 0x8003, 0xea gets loaded.

"So what the hell is really going on?"

This set of commands simply executes an infinite loop.

Review the datasheet:

https://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

The above code has 3 parts to it:

1)Conditional – Set To Always

2)Op Code – Branch

3)Offset – How Far To Move Within The Current Location

Condition – bits 31-28: 0xe or 1110

Op Code – bits 27-24: 0xa or 1010

Offset – bits 23-0 -2

I know this may be a lot to wrap your mind around however it is critical that you take the time and read the datasheet linked above. Do not cut corners if you truly have the passion to understand the above. READ THE DATASHEET!

I will go through painstaking efforts to break everything down step-by-step however there are exercises like the above that I am asking you to review the datasheet above so you learn how to better understand where to look when you are stuck on a particular routine or set of machine code. This is one of those times I ask you to please read and research the datasheet above!

“I’m bored! Why the hell does this crap matter?”

Glad you asked! The single most dangerous malware on planet earth today is that of the root-kit variety. If you do not have a basic understanding of the above, you will never begin to even understand what a root-kit is as you progress in your understanding.

Anyone can simply replace the **kernel.img** file with their own hacked version and you can have total control over the entire process from boot.

Next week we will dive into the Von Neumann Architecture.

Part 14 - Hello World

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Today we begin our journey into the world of C++ and gaining a better understanding of how C++ interacts with our ARM processor.

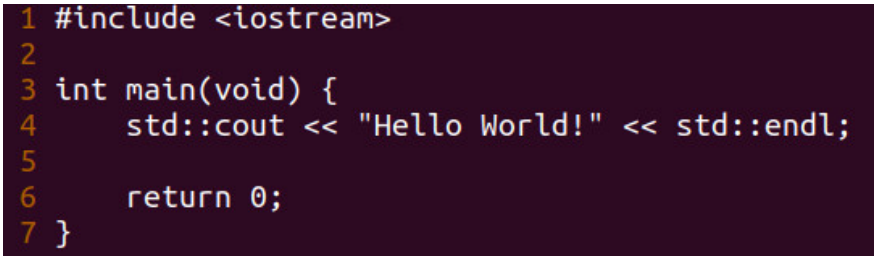
The prior lessons in this series focus on the basics of the ARM processor and touch upon its architecture and how everything ultimately translates down to Assembly Language and then ultimately opcodes into machine language.

We start with our first program in C++ which is our “Hello World” program. Let’s dive in and break each line down step-by-step and see how this language works. We will call this **example1.cpp** and save it to our device.

```
#include <iostream>

int main(void) {
    std::cout << “Hello World” std::endl;

    return 0;
}
```

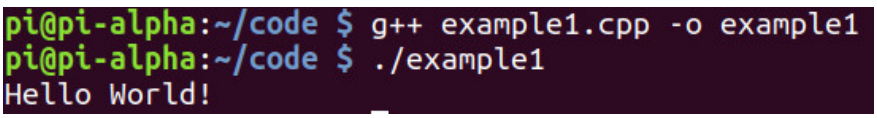


To compile this we simply type:

```
g++ example1.cpp -o example1
```

We simply then type:

```
./example1
```



SUCCESS! We see “Hello World” printed to the standard output or terminal!

Lets break it down line by line:

#include <iostream> is referred to as a preprocessor statement. These preprocessor statements happen just before the compilation of the rest of the code. The #include keyword will find a file called **iostream** and take all of the contents of that file and paste it into the existing code we just created. These files are also called header files.

We call **iostream** because we need a declaration for a function called **cout** and **endl**. The **cout** function allows us to print text to the standard output or terminal and the **endl** function creates a new line after the text has been displayed.

The main section which is of type integer is the entry point into the main application or binary. You will notice a **void** inside the **()** which indicates that it does not have any parameters which will be passed into the function.

The **std** indicates a namespace which is quite simply a mechanism to organize code into logical groups in order to prevent name collisions when you are dealing with multiple libraries.

You will see many examples where they declare a using namespace std; however I will NEVER utilize this approach as it can cause naming collisions in more complex applications.

The **<<** operator is referred to as an overloaded operator. They are essentially a function very similar to **printf** in the C language. We are simply moving the **“Hello World”** string into the **cout** function through the use of the **<<** overloaded operator. We then push the **endl** which creates a new line to the console.

The final line is the return 0. Since our main function is of type int, we have to return something. In C++ 11 there is no need for this in the main function however is required for every other function. I will stick to tradition and simply include it.

The next stage is that we compile the file. The first thing that occurs is the entire contents of the iostream header goes into the source file as we discussed. The compile process is where the C++ code gets translated into machine code. The next stage of compilation occurs when the rest of the lines of our existing code are parsed through. Essentially we have all of the contents of iostream into a new file and then all of the contents of our existing file added to a single file.

Compiling takes our text file the cpp file and converts it into an intermediate format called an obj file. An abstract syntax tree is created which is a conversion of constant data, variables and instructions.

Once the tree is created the code is generated. This means we now have machine code that our ARM CPU will execute. Every cpp file (translation units) which will have its own respective obj file associated with it.

Linking takes our obj files, our compiled files, in addition to the C++ Standard Library and finds where each symbol and function is and link them all together into one executable.

The concepts above may appear a bit confusing if you are new to programming however as you code and compile and later debug and hack in Assembly Language it will all become very clear and you will learn to master the processor.

Next week we will dive into Debugging Hello World.

Part 15 - Debugging Hello World

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code from last week.

```
1 #include <iostream>
2
3 int main(void) {
4     std::cout << "Hello World!" << std::endl;
5
6     return 0;
7 }
```

Let's debug! Let's fire up GDB which is the GNU Debugger to which we will break down the C++ binary and step through it line-by-line in ARM Assembly.

```
pi@pi-alpha:~/code $ gdb -q example1
Reading symbols from example1...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:     push    {r11, lr}
   0x00010720 <+4>:     add     r11, sp, #4
   0x00010724 <+8>:     ldr     r0, [pc, #32]   ; 0x1074c <main+48>
   0x00010728 <+12>:    ldr     r1, [pc, #32]   ; 0x10750 <main+52>
   0x0001072c <+16>:    bl      0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:    mov     r3, r0
   0x00010734 <+24>:    mov     r0, r3
   0x00010738 <+28>:    ldr     r1, [pc, #20]   ; 0x10754 <main+56>
   0x0001073c <+32>:    bl      0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:    mov     r3, #0
   0x00010744 <+40>:    mov     r0, r3
   0x00010748 <+44>:    pop     {r11, pc}
   0x0001074c <+48>:    ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010750 <+52>:    andeq   r0, r1, r8, asr #16
   0x00010754 <+56>:    andeq   r0, r1, r8, ror #11
End of assembler dump.
```

This is the ARM disassembly that we are seeing. No matter what language you program in, it ultimately will go down to this level.

This might be a bit scary to you if you did not take my prior course on ARM Assembly. If you need to do a refresher, please link back to that series.

You are probably asking yourself why we are not debugging with the original source code and seeing how it matches nicely to the assembly. The answer is when you are a professional Reverse Engineer, you do not get the luxury of seeing source code when you are reversing binaries.

This is a childishly simple example and we will continue through the series with very simple examples so that you can learn effective techniques. We are using a text-based debugger here so that you fully understand what is going on and to

also get some training if you had to ever attach yourself to a running process inside a foreign machine you will know how to properly debug or hack.

I will focus SOLELY on this method rather than using a nice graphical debugger like IDA or the like so that you are able to manipulate at a very low-level.

We start with loading the link register into **r11** and adding **4** to the stack pointer and then adding it to **r11**. This is simply a routine which will allow the binary to preserve the link register and setting up space on the stack.

We notice memory address **0x10750** being loaded from memory to the register **r1**. Let's do a string examination and see what is located at that address.

```
(gdb) x/s *0x10750
0x10848:        "Hello World!"
```

Voila! We see our string. **"Hello World!"** located at that memory address.

Let's set a breakpoint at **main+16.**

```
(gdb) b *main+16
Breakpoint 2 at 0x1072c
(gdb) s
Single stepping until exit from function main,
which has no line number information.

Breakpoint 2, 0x0001072c in main ()
(gdb) disas
Dump of assembler code for function main:
   0x0001071c <+0>:     push    {r11, lr}
   0x00010720 <+4>:     add     r11, sp, #4
   0x00010724 <+8>:     ldr     r0, [pc, #32]   ; 0x1074c <main+48>
   0x00010728 <+12>:    ldr     r1, [pc, #32]   ; 0x10750 <main+52>
=> 0x0001072c <+16>:    bl      0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:    mov     r3, r0
   0x00010734 <+24>:    mov     r0, r3
   0x00010738 <+28>:    ldr     r1, [pc, #20]   ; 0x10754 <main+56>
   0x0001073c <+32>:    bl      0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:    mov     r3, #0
   0x00010744 <+40>:    mov     r0, r3
   0x00010748 <+44>:    pop     {r11, pc}
   0x0001074c <+48>:    ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010750 <+52>:    andeq   r0, r1, r8, asr #16
   0x00010754 <+56>:    andeq   r0, r1, r8, ror #11
End of assembler dump.
```

Let's take a look at our register values.

```
(gdb) i r
r0             0x209d0  133584
r1             0x10848  67656
r2             0x7efff39c       2130703260
r3             0x1071c  67356
r4             0x0      0
r5             0x0      0
r6             0x105f4  67060
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x76fff000       1996484608
r11            0x7efff23c       2130702908
r12            0x76e1f000       1994518528
sp             0x7efff238       0x7efff238
lr             0x76cfa294       1993319060
pc             0x1072c  0x1072c <main+16>
cpsr           0x60000010       1610612752
(gdb) x/s $r1
0x10848:        "Hello World!"
```

Let's now take a look at what is inside the **r1** register and then step through the binary.

```
(gdb) x/s $r1
0x10848:        "Hello World!"
(gdb) s
Single stepping until exit from function main,
which has no line number information.
Hello World!
__libc_start_main (main=0x7efff394, argc=1994518528,
    argv=0x76cfa294 <__libc_start_main+276>, init=<optimized out>,
    fini=0x10838 <__libc_csu_fini>, rtld_fini=0x76fdfa14 <_dl_fini>,
    stack_end=0x7efff394) at libc-start.c:321
321     libc-start.c: No such file or directory.
```

We see the **“Hello World!”** string now residing inside of **r1** which resides at memory address **0x10848**. Finally let’s continue through the binary.

```
(gdb) c
Continuing.
[Inferior 1 (process 1069) exited normally]
```

Understanding assembly and step-by-step debugging allows you to have complete and ultimate control over any binary! More complex binaries can cause you hours, days or weeks to truly Reverse Engineer however the techniques are the same just more time consuming.

Reverse Engineering is the most sophisticated form of analysis in advanced Computer Engineering. There are many tools that a professional Reverse Engineer uses however each of those tools have a usage and purpose however this technique is the most sophisticated and comprehensive.

Next week we will dive into Hacking Hello World.

Part 16 - Hacking Hello World

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code from two weeks ago.

```
1 #include <iostream>
2
3 int main(void) {
4     std::cout << "Hello World!" << std::endl;
5
6     return 0;
7 }
```

Let's debug once again.

```
pi@pi-alpha:~/code $ gdb -q example1
Reading symbols from example1...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:     push    {r11, lr}
   0x00010720 <+4>:     add     r11, sp, #4
   0x00010724 <+8>:     ldr     r0, [pc, #32]   ; 0x1074c <main+48>
   0x00010728 <+12>:    ldr     r1, [pc, #32]   ; 0x10750 <main+52>
   0x0001072c <+16>:    bl      0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:    mov     r3, r0
   0x00010734 <+24>:    mov     r0, r3
   0x00010738 <+28>:    ldr     r1, [pc, #20]   ; 0x10754 <main+56>
   0x0001073c <+32>:    bl      0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:    mov     r3, #0
   0x00010744 <+40>:    mov     r0, r3
   0x00010748 <+44>:    pop     {r11, pc}
   0x0001074c <+48>:    ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010750 <+52>:    andeq   r0, r1, r8, asr #16
   0x00010754 <+56>:    andeq   r0, r1, r8, ror #11
End of assembler dump.
```

Let's once again examine the contents of the string at memory address **0x10750** and continue through the execution of the program.

```
(gdb) x/s *0x10750
0x10848:        "Hello World!"
(gdb) c
Continuing.
Hello World!
[Inferior 1 (process 1038) exited normally]
```

As you can see it holds the "**Hello World!**" string and when we continue through it echo's back to the terminal as such.

Let's hack! Let's now overwrite the value inside of the memory address with the string, "**Hacked World!**" and continue execution.

```
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) x/s *0x10750
0x10848:        "Hello World!"
(gdb) set *0x10750 = "Hacked World!"
(gdb) c
Continuing.
Hacked World!
[Inferior 1 (process 1045) exited normally]
```

Woohoo! Our first hack! As you can see as you understand Assembly you have ABSOLUTE control over the entire binary no matter what language it is written in. In this very simple example we were able to hack the value inside the memory address of **0x10750** to which when executed it echoed, “**Hacked World!**” to the terminal or standard output.

Let’s again run the binary and do a disassembly.

```
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:     push    {r11, lr}
   0x00010720 <+4>:     add     r11, sp, #4
   0x00010724 <+8>:     ldr     r0, [pc, #32]   ; 0x1074c <main+48>
   0x00010728 <+12>:    ldr     r1, [pc, #32]   ; 0x10750 <main+52>
   0x0001072c <+16>:    bl      0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:    mov     r3, r0
   0x00010734 <+24>:    mov     r0, r3
   0x00010738 <+28>:    ldr     r1, [pc, #20]   ; 0x10754 <main+56>
   0x0001073c <+32>:    bl      0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:    mov     r3, #0
   0x00010744 <+40>:    mov     r0, r3
   0x00010748 <+44>:    pop     {r11, pc}
   0x0001074c <+48>:    ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010750 <+52>:    andeq   r0, r1, r8, asr #16
   0x00010754 <+56>:    andeq   r0, r1, r8, ror #11
End of assembler dump.
```

Let’s now do the same procedure however lets **si** 3x and examine the string inside of **r1**. We see that it contains, “**Hello World!**” as it has been successfully **ldr** (load from memory into the register) at **main+12**.

Let’s now set r1 to “**Hacked World!**” and continue execution. As you can see we now hacked it coming out of the register rather than in memory. You can clearly begin to see there are a number of ways to hack anything and here is a simple example of two such ways.

```
(gdb) si
0x00010720 in main ()
(gdb) si
0x00010724 in main ()
(gdb) si
0x00010728 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x0001071c <+0>:     push    {r11, lr}
   0x00010720 <+4>:     add     r11, sp, #4
   0x00010724 <+8>:     ldr     r0, [pc, #32]   ; 0x1074c <main+48>
=> 0x00010728 <+12>:    ldr     r1, [pc, #32]   ; 0x10750 <main+52>
   0x0001072c <+16>:    bl      0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:    mov     r3, r0
   0x00010734 <+24>:    mov     r0, r3
   0x00010738 <+28>:    ldr     r1, [pc, #20]   ; 0x10754 <main+56>
   0x0001073c <+32>:    bl      0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:    mov     r3, #0
   0x00010744 <+40>:    mov     r0, r3
   0x00010748 <+44>:    pop     {r11, pc}
   0x0001074c <+48>:    ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010750 <+52>:    andeq   r0, r1, r8, asr #16
   0x00010754 <+56>:    andeq   r0, r1, r8, ror #11
End of assembler dump.
(gdb) si
0x0001072c in main ()
(gdb) x/s $r1
0x10848:        "Hello World!"
(gdb) set $r1 = "Hacked World!"
(gdb) c
Continuing.
Hacked World!
[Inferior 1 (process 1050) exited normally]
```

Reverse Engineering is all about understanding how a program executes and hijacking execution flow and changing values to suit our purpose! Today you took your first step into this amazing journey!

Next week we will dive into constants.

Part 17 - Constants

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

So far we have created, debugged and hacked a simple string echo to the standard terminal. We will expand upon that example by adding a constant.

A constant in C++ is a value that will not change throughout program execution (unless hacked). It is used such that you have a declaration early in the code so that if your future program architecture ever changes you can redefine the constant in one place rather than having to update code all through your code base.

It is standard practice to code our constants in all CAPS so that when we see it referenced somewhere in the code we know that value is a constant.

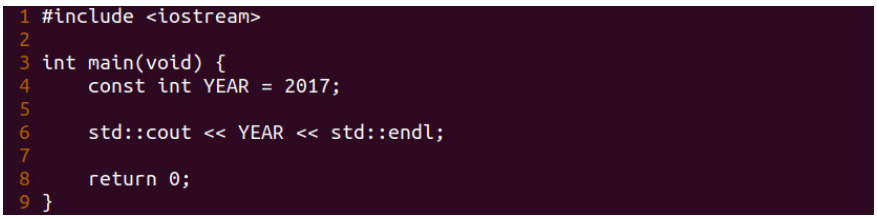
We start with our second program in C++ which is our “Constant” program. Let’s dive in and break each line down step-by-step and see how this language works. We will call this **example2.cpp** and save it to our device.

```
#include <iostream>

int main(void) {
            cons tint YEAR = 2017;

            std::cout << YEAR << std::endl;

            return 0;
}
```



To compile this we simply type:

```
g++ example2.cpp -o example2
```

We simply then type:

```
./example2
```



SUCCESS! We see “**2017**” printed to the standard output or terminal!

Let’s break it down:

We utilize the **const** keyword to indicate a constant to which we assign it the integer value of 2017.

We then utilize the **cout** function to print it to the standard output or terminal and add a new line with the **endl** function.

That’s it! Very simple.

Next week we will dive into Debugging Constants.

Part 18 – Debugging Constants

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review last week's code.

```
#include <iostream>

int main(void) {
    const int YEAR = 2017;

    std::cout << YEAR << std::endl;

    return 0;
}
```

Let's debug!

```
pi@pi-alpha:~/code $ gdb -q example2
Reading symbols from example2...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example2

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    ldr     r1, [pc, #32]   ; 0x10730 <main+64>
   0x0001070c <+28>:    bl      0x1055c
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    andeq   r0, r0, r1, ror #15
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print *0x10730
$1 = 2017
```

As we can see the value in the memory address **0x10730** is equal to **2017**. Let's continue and watch the value print to the standard output (terminal) as it did last week when we ran it.

```
(gdb) c
Continuing.
2017
[Inferior 1 (process 1008) exited normally]
```

We can see very clearly that we move the value from memory into **r1** and then we branch to our **cout** function to print to the terminal. At this stage you should feel a little more comfortable with understanding what the assembly is doing above.

Next week we will dive into Hacking Constants.

Part 19 – Hacking Constants

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our original code.

```
#include <iostream>

int main(void) {
    const int YEAR = 2017;

    std::cout << YEAR << std::endl;

    return 0;
}
```

Let's hack!

```
pi@pi-alpha:~/code $ gdb -q example2
Reading symbols from example2...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example2

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    ldr     r1, [pc, #32]   ; 0x10730 <main+64>
   0x0001070c <+28>:    bl      0x1055c
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    andeq   r0, r0, r1, ror #15
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print *0x10730
$1 = 2017
(gdb) set *0x10730 = 1981
(gdb) c
Continuing.
1981
[Inferior 1 (process 1046) exited normally]
```

As we can see the value in the memory address **0x10730** is equal to **2017**. Let's change that value in memory to **1981**. Let's continue and watch the value turn to **1981**! Successful hack!

Let's hack a second way! Re-start the program and set a breakpoint at main+28 and continue to the breakpoint.

```
(gdb) r
Starting program: /home/pi/code/example2

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    ldr     r1, [pc, #32]   ; 0x10730 <main+64>
   0x0001070c <+28>:    bl      0x1055c
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    andeq   r0, r0, r1, ror #15
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

Let's continue and we see the value in **r1** is **2017**. Let's change the value in **r1** to **1981**. We continue and see the program successfully hacked to **1981**!

```
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    ldr     r1, [pc, #32]   ; 0x10730 <main+64>
=> 0x0001070c <+28>:    bl      0x1055c
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    andeq   r0, r0, r1, ror #15
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print $r1
$2 = 2017
(gdb) set $r1 = 1981
(gdb) c
Continuing.
1981
[Inferior 1 (process 1049) exited normally]
```

Next week we will dive into Character Variables.

Part 20 – Character Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The next stage in our journey is that of character variables. Unlike the strings we have dealt with thus far, a character only takes up one byte of data.

Keep in mind, when we deal with any character data, we deal with literally two hex digits which are the ASCII code that represents an actual character that we see on our respective terminals.

Remember that each hex digit is 4 bits in length. Therefore two hex digits are 8 bits in length or a byte long.

To recap, each character translates down to an ASCII code in hex which the processor understands. The value of **n** is **0x6e** hex or **110** decimal. You can review any ASCII table to see where we derived this value. This will come in handy in the next lesson.

We start with our third program in C++ which is our “Character Variable” program. Let’s dive in and break each line down step-by-step and see how this language works. We will call this example3.cpp and save it to our device.

```
#include <iostream>



int main(void) {

            char yes_no = ‘n’;



            std::cout << yes_no << std::endl;



            return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     char yes_no = 'n';
5
6     std::cout << yes_no << std::endl;
7
8     return 0;
9 }
```

To compile this we simply type:

g++ example3.cpp -o example3

We simply then type:

./example3

```
pi@pi-alpha:~/code $ g++ example3.cpp -o example3
pi@pi-alpha:~/code $ ./example3
n
```

SUCCESS! We see “**n**” printed to the standard output or terminal!

Let’s break it down:

We utilize the **char** keyword to indicate a character variable to which we assign it the value of **n**.

We then utilize the **cout** function to print it to the standard output or terminal and add a new line with the **endl** function.

That’s it! Very simple.

Next week we will dive into Debugging Character Variables.

Part 21 – Debugging Character Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code.

```
#include <iostream>

int main(void) {
    char yes_no = 'n';

    std::cout << yes_no << std::endl;

    return 0;
}
```

Let's debug!

```
pi@pi-alpha:~/code $ gdb -q example3
Reading symbols from example3...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example3

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:     push    {r11, lr}
   0x00010720 <+4>:     add     r11, sp, #4
   0x00010724 <+8>:     sub     sp, sp, #8
   0x00010728 <+12>:    mov     r3, #110        ; 0x6e
   0x0001072c <+16>:    strb    r3, [r11, #-5]
   0x00010730 <+20>:    ldrb    r3, [r11, #-5]
   0x00010734 <+24>:    ldr     r0, [pc, #36]   ; 0x10760 <main+68>
   0x00010738 <+28>:    mov     r1, r3
   0x0001073c <+32>:    bl      0x105b8
   0x00010740 <+36>:    mov     r3, r0
   0x00010744 <+40>:    mov     r0, r3
   0x00010748 <+44>:    ldr     r1, [pc, #20]   ; 0x10764 <main+72>
   0x0001074c <+48>:    bl      0x105dc <_ZNSt8ios_base4InitD1Ev+24>
   0x00010750 <+52>:    mov     r3, #0
   0x00010754 <+56>:    mov     r0, r3
   0x00010758 <+60>:    sub     sp, r11, #4
   0x0001075c <+64>:    pop     {r11, pc}
   0x00010760 <+68>:    ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010764 <+72>:    andeq   r0, r1, r8, ror #11
End of assembler dump.
```

Woah! This is confusing. I don't see any clear memory addresses being loaded into a register to manipulate the data.

Let's keep in mind that we are dealing with a single byte character variable.

If you remember from last week each character translates down to an ASCII code in hex which the processor understands. The value of **n** is **0x6e** hex or **110** decimal. You can review any ASCII table to see where we derived this value.

We do see **0x6e** at **main+12** which is the character '**n**'.

```
(gdb) si
0x00010720 in main ()
(gdb) si
0x00010724 in main ()
(gdb) si
0x00010728 in main ()
(gdb) si
0x0001072c in main ()
(gdb) i r
r0             0x1      1
r1             0x7efff394       2130703252
r2             0x7efff39c       2130703260
r3             0x6e     110
r4             0x0      0
r5             0x0      0
r6             0x105f4  67060
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x76fff000       1996484608
r11            0x7efff23c       2130702908
r12            0x76e1f000       1994518528
sp             0x7efff230       0x7efff230
lr             0x76cfa294       1993319060
pc             0x1072c  0x1072c <main+16>
cpsr           0x60000010       1610612752
(gdb) print/c $r3
$1 = 110 'n'
```

If we step into a few times we notice the value has been placed into **r3**. When we print the value in **r3** we now see our '**n**' character.

Let's continue.

```
(gdb) c
Continuing.
n
[Inferior 1 (process 1567) exited normally]
```

We now see the '**n**' printed to the standard output as expected.

It is important that you understand this process and understand that each character translates into an ASCII value to which the processor loads directly into a respective register. Our previous experience we have seen a string loaded directly into a memory location and this is not the case here.

Next week we will dive into Hacking Character Variables.

Part 22 – Hacking Character Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code.

```
#include <iostream>

int main(void) {
    char yes_no = 'n';

    std::cout << yes_no << std::endl;

    return 0;
}
```

Let's hack!

```
pi@pi-alpha:~/code $ gdb -q example3
Reading symbols from example3...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example3

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:     push    {r11, lr}
   0x00010720 <+4>:     add     r11, sp, #4
   0x00010724 <+8>:     sub     sp, sp, #8
   0x00010728 <+12>:    mov     r3, #110        ; 0x6e
   0x0001072c <+16>:    strb    r3, [r11, #-5]
   0x00010730 <+20>:    ldrb    r3, [r11, #-5]
   0x00010734 <+24>:    ldr     r0, [pc, #36]   ; 0x10760 <main+68>
   0x00010738 <+28>:    mov     r1, r3
   0x0001073c <+32>:    bl      0x105b8
   0x00010740 <+36>:    mov     r3, r0
   0x00010744 <+40>:    mov     r0, r3
   0x00010748 <+44>:    ldr     r1, [pc, #20]   ; 0x10764 <main+72>
   0x0001074c <+48>:    bl      0x105dc <_ZNSt8ios_base4InitD1Ev+24>
   0x00010750 <+52>:    mov     r3, #0
   0x00010754 <+56>:    mov     r0, r3
   0x00010758 <+60>:    sub     sp, r11, #4
   0x0001075c <+64>:    pop     {r11, pc}
   0x00010760 <+68>:    ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010764 <+72>:    andeq   r0, r1, r8, ror #11
End of assembler dump.
```

We again see the direct value of **0x6e** moved into **r3** at **main+12** which is our '**n**'.

```
(gdb) si
0x00010720 in main ()
(gdb) si
0x00010724 in main ()
(gdb) si
0x00010728 in main ()
(gdb) si
0x0001072c in main ()
(gdb) print/c $r3
$1 = 110 'n'
```

After stepping into 4 times and verify the value in **r3** which we clearly see as '**n**'.

```
(gdb) set $r3 = 'y'
(gdb) print/c $r3
$2 = 121 'y'
```

Let's hack the value in **r3** to a '**y**' and then reexamine the value in **r3**. We can now clearly see it has been changed to '**y**'.

```
(gdb) c
Continuing.
y
[Inferior 1 (process 1587) exited normally]
```

As we continue we successfully see our hack worked! We see the value of '**y**' printing to the standard output.

Next week we will dive into Boolean Variables.

Part 23 – Boolean Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The next stage in our journey is that of Boolean variables. The name goes back to the great George Boole to which all modern computer science has derived.

At the lowest level a value is either 0 or 1, false or true, + < 5 volts or +5 volts, etc.

Let's examine our code.

```
#include <iostream>



int main(void) {

            bool isHacked = false;

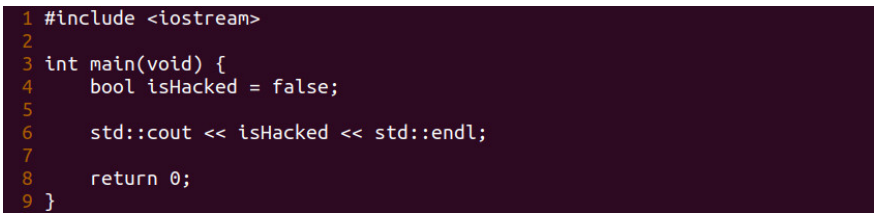


            std::cout << isHacked << std::endl;



            return 0;

}
```



```
#include <iostream>

int main(void) {
    bool isHacked = false;

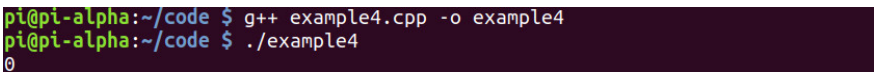
    std::cout << isHacked << std::endl;

    return 0;
}
```

To compile this we simply type:

```
g++ example4.cpp -o example4


./example4
```



```
pi@pi-alpha:~/code $ g++ example4.cpp -o example4
pi@pi-alpha:~/code $ ./example4
0
```

SUCCESS! We see **0** printed to the standard output or terminal!

Let's break it down:

We create a boolean variable called **isHacked** to which we assign a value of **false** or **0**. When we run the binary we clearly see the value **0** that successfully was echoed to the standard output.

Next week we will dive into Debugging Boolean Variables.

Part 24 – Debugging Boolean Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>

int main(void) {
    bool isHacked = false;

    std::cout << isHacked << std::endl;

    return 0;
}
```

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q example4
Reading symbols from example4...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example4

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #0
   0x00010700 <+16>:    strb    r3, [r11, #-5]
   0x00010704 <+20>:    ldrb    r3, [r11, #-5]
   0x00010708 <+24>:    ldr     r0, [pc, #36]   ; 0x10734 <main+68>
   0x0001070c <+28>:    mov     r1, r3
   0x00010710 <+32>:    bl      0x10598 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #20]   ; 0x10738 <main+72>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's step 4 times and disassemble.

```
(gdb) si
0x000106f4 in main ()
(gdb) si
0x000106f8 in main ()
(gdb) si
0x000106fc in main ()
(gdb) si
0x00010700 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #0
=> 0x00010700 <+16>:    strb    r3, [r11, #-5]
   0x00010704 <+20>:    ldrb    r3, [r11, #-5]
   0x00010708 <+24>:    ldr     r0, [pc, #36]   ; 0x10734 <main+68>
   0x0001070c <+28>:    mov     r1, r3
   0x00010710 <+32>:    bl      0x10598 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #20]   ; 0x10738 <main+72>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's examine what is now in **r3**.

```
(gdb) print $r3
$1 = 0
(gdb) c
Continuing.
0
[Inferior 1 (process 21451) exited normally]
```

As we can clearly see the value in **isHacked** is **0** or **false** which makes sense based on our c++ source code.

I know these lessons may seem trivial however Reverse Engineering is all about breaking things down in their most basic components. Reverse Engineering is about patience and logical flow. It is critical that you take the time and work through all of these examples with a Raspberry Pi device so that you can have a proper appreciation for how the process actually works.

Next week we will dive into Hacking Boolean Variables.

Part 25 – Hacking Boolean Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>

int main(void) {
    bool isHacked = false;

    std::cout << isHacked << std::endl;

    return 0;
}
```

Let's hack!

```
pi@pi-alpha:~/code $ gdb -q example4
Reading symbols from example4...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example4
disas
Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #0
   0x00010700 <+16>:    strb    r3, [r11, #-5]
   0x00010704 <+20>:    ldrb    r3, [r11, #-5]
   0x00010708 <+24>:    ldr     r0, [pc, #36]   ; 0x10734 <main+68>
   0x0001070c <+28>:    mov     r1, r3
   0x00010710 <+32>:    bl      0x10598 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #20]   ; 0x10738 <main+72>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) si
0x000106f4 in main ()
(gdb) si
0x000106f8 in main ()
(gdb) si
0x000106fc in main ()
(gdb) si
0x00010700 in main ()
```

Let's break at main, run and disas in addition to step into four times.

```
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #0
=> 0x00010700 <+16>:    strb    r3, [r11, #-5]
   0x00010704 <+20>:    ldrb    r3, [r11, #-5]
   0x00010708 <+24>:    ldr     r0, [pc, #36]   ; 0x10734 <main+68>
   0x0001070c <+28>:    mov     r1, r3
   0x00010710 <+32>:    bl      0x10598 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #20]   ; 0x10738 <main+72>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We see that **0** or **FALSE** is moved into **r3** at main+12.

```
(gdb) set $r3 = 1
(gdb) print $r3
$1 = 1
(gdb) c
Continuing.
1
[Inferior 1 (process 21457) exited normally]
```

Very simply we set **r3** to **1** or **TRUE** and continue execution to which we notice that the Boolean variable **isHacked** is now **TRUE**.

It’s that simple folks! These elementary examples will help build your mental library of examples of how to approach everything in code and understanding how to take control of code execution no matter what!

Next week we will dive into Integer Variables.

Part 26 – Integer Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The next stage in our journey is that of Integer variables.

A 32-bit register can store 2^32 different values. The range of integer values that can be stored in 32 bits depends on the integer representation used. With the two most common representations, the range is 0 through 4,294,967,295 (2^32 − 1) for representation as an (unsigned) binary number, and −2,147,483,648 (−2^31) through 2,147,483,647 (2^31 − 1) for representation as two's complement.

Keep in mind with 32-bit memory addresses you can directly access a maximum of 4 GB of byte-addressable memory.

Let's examine our code.

```
#include <iostream>



int main(void) {

            int myNumber = 777;



            std::cout << myNumber << std::endl;



            return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     int myNumber = 777;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

To compile this we simply type:

g++ example5.cpp -o example5

./example5

```
pi@pi-alpha:~/code $ g++ example5.cpp -o example5
pi@pi-alpha:~/code $ ./example5
777
```

SUCCESS! We see **777** printed to the standard output or terminal!

Let’s break it down:

We assign the integer **777** directly into the variable **myNumber** and then print it out to the terminal with the c++ **cout** function.

Next week we will dive into Debugging Integer Variables.

Part 27 – Debugging Integer Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code. I again want to include the below information from last week's lesson to emphasize what is going on regarding integers.

A 32-bit register can store 2^32 different values. The range of integer values that can be stored in 32 bits depends on the integer representation used. With the two most common representations, the range is 0 through 4,294,967,295 (2^32 − 1) for representation as an (unsigned) binary number, and −2,147,483,648 (−2^31) through 2,147,483,647 (2^31 − 1) for representation as two's complement.

Keep in mind with 32-bit memory addresses you can directly access a maximum of 4 GB of byte-addressable memory.

```
#include <iostream>

int main(void) {
    int myNumber = 777;

    std::cout << myNumber << std::endl;

    return 0;
}
```

Let's debug!

```
pi@pi-alpha:~/code $ gdb -q example5
Reading symbols from example5...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example5

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    ldr     r1, [r11, #-8]
   0x0001070c <+28>:    bl      0x1055c
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    andeq   r0, r0, r9, lsl #6
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We see at **main+12** the address at **0x10730** loading data into **r3**. Let's take a closer look.

```
(gdb) x/d 0x10730
0x10730 <main+64>:      777
(gdb) c
Continuing.
777
[Inferior 1 (process 1141) exited normally]
```

When we examine the data inside **0x10730** we clearly see the integer **777** present. When we continue we see **777** echoed back to the terminal which makes sense as we utilized the **cout** function within c++.#linux #arm #asm #cplusplus #reverseengineering

Next week we will dive into Hacking Integer Variables.

Part 28 – Hacking Integer Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code.

```
 1 #include <iostream>
 2
 3 int main(void) {
 4     int myNumber = 777;
 5
 6     std::cout << myNumber << std::endl;
 7
 8     return 0;
 9 }
```

Let's hack!

```
pi@pi-alpha:~/code $ gdb -q example5
Reading symbols from example5...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example5

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    ldr     r1, [r11, #-8]
   0x0001070c <+28>:    bl      0x1055c
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    andeq   r0, r0, r9, lsl #6
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's take a look again inside the memory location of **0x10730**.

```
(gdb) x/d 0x10730
0x10730 <main+64>:      777
(gdb) c
Continuing.
777
[Inferior 1 (process 1141) exited normally]
```

As we can clearly see the integer value of **777** appears and when we continue it echoes out to the terminal the value of **777** which corresponds with our c++ function **cout**.

Let's hack the value inside of **0x10730** and set the value to **666** and then reexamine the value inside **0x10730** and continue.

```
(gdb) x/d 0x10730
0x10730 <main+64>:      777
(gdb) set *0x10730 = 666
(gdb) x/d 0x10730
0x10730 <main+64>:      666
(gdb) c
Continuing.
666
[Inferior 1 (process 1145) exited normally]
```

Success! As we can see we hacked the value to **666** as we continue we see it echoed out to stdout.

Next week we will dive into Float Variables.

Part 29 – Float Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The next stage in our journey is that of Floating-Point variables.

A floating-point variable is different from an integer as it has a fractional value attached to which we designate with a period.

Let's examine our code.

```
#include <iostream>



float main(void) {

            int myNumber = 1337.1;

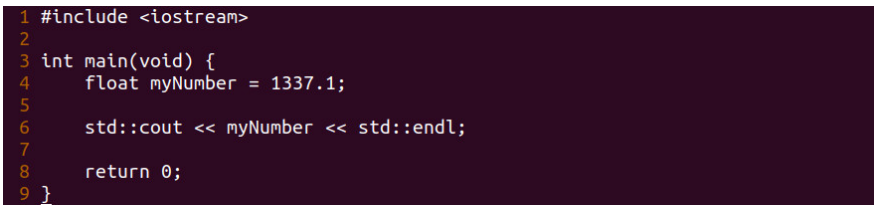


            std::cout << myNumber << std::endl;



            return 0;

}
```



To compile this we simply type:

g++ example6.cpp -o example6

./example6

SUCCESS! We see 1337.1 printed to the standard output or terminal!

Let's break it down:

We assign the floating-point variable directly into the variable **myNumber** and then print it out to the terminal with the c++ **cout** function.

Thus far we have a good understanding of the ARM registers however next week we will introduce the registers within the math co-processor that work with floating-point variables. The registers you have worked with up to now only store whole numbers or integers and at the Assembly level, any fractional value must be manipulated through the math co-processor registers.

Next week we will dive into Debugging Float Variables.

Part 30 – Debugging Float Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>



int main(void) {

            float myNumber = 1337.1;

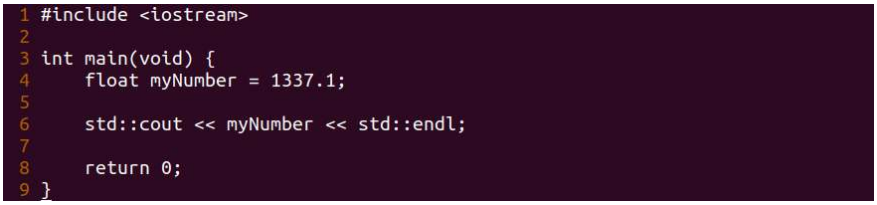


            std::cout << myNumber << std::endl;



            return 0;

}
```



Let's debug!

```
pi@pi-alpha:~/code $ gdb -q example6
Reading symbols from example6...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example6
disas
Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    vldr    s0, [r11, #-8]
   0x0001070c <+28>:    bl      0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    strtmi  r2, [r7], #819  ; 0x333
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break on **main+20** and continue to that point.

```
(gdb) b *main+20
Breakpoint 2 at 0x10704
(gdb) c
Continuing.

Breakpoint 2, 0x00010704 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
=> 0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    vldr    s0, [r11, #-8]
   0x0001070c <+28>:    bl      0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    strtmi  r2, [r7], #819  ; 0x333
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's examine what value is inside **r11-8**. We clearly see it is **1337.09998** which approximates our value in our original c++ code. Keep in mind a float has roughly 7 decimal digits of precision and that is why we do not see **1337.1** so please remember that as we go forward.

```
(gdb) x/f $r11-8
0x7efff234:     1337.09998
```

We can also see this value in high memory.

```
(gdb) x/f 0x7efff234
0x7efff234:     1337.09998
```

Let's break on **main+28** and continue.

```
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

We see a strange new instruction. We see **vldr** and the value within **r11, #8** being moved into **s0**. So what is **s0**? We have a math co-processor which has a series of additional registers that work with decimal or floating-point numbers. Here we see an example of such to which the value of **1337.09998** is being moved into **s0**. The **vldr** instruction loads a constant value into every element of a single-precision or double-precision register such as **s0**.

```
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    vldr    s0, [r11, #-8]
=> 0x0001070c <+28>:    bl      0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    strtmi  r2, [r7], #819  ; 0x333
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We can only see these special registers if we do a info registers all command as we do below.

```
(gdb) i r a
```

Below we see the value now being moved into **s0**.

```
s0             1337.09998       (raw 0x44a72333)
s1             0        (raw 0x00000000)
s2             0        (raw 0x00000000)
s3             0        (raw 0x00000000)
s4             0        (raw 0x00000000)
s5             0        (raw 0x00000000)
s6             0        (raw 0x00000000)
s7             0        (raw 0x00000000)
s8             0        (raw 0x00000000)
s9             0        (raw 0x00000000)
s10            0        (raw 0x00000000)
s11            0        (raw 0x00000000)
s12            0        (raw 0x00000000)
s13            2        (raw 0x40000000)
s14            0        (raw 0x00000000)
s15            0        (raw 0x00000000)
s16            0        (raw 0x00000000)
s17            0        (raw 0x00000000)
s18            0        (raw 0x00000000)
s19            0        (raw 0x00000000)
s20            0        (raw 0x00000000)
s21            0        (raw 0x00000000)
s22            0        (raw 0x00000000)
s23            0        (raw 0x00000000)
s24            0        (raw 0x00000000)
s25            0        (raw 0x00000000)
s26            0        (raw 0x00000000)
s27            0        (raw 0x00000000)
s28            0        (raw 0x00000000)
s29            0        (raw 0x00000000)
s30            0        (raw 0x00000000)
s31            0        (raw 0x00000000)
```

Next week we will dive into Hacking Float Variables.

Part 31 – Hacking Float Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>



int main(void) {

            int myNumber = 1337.1;

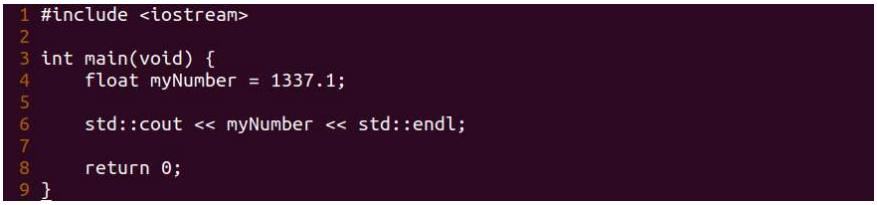


            std::cout << myNumber << std::endl;



            return 0;

}
```



```
#include <iostream>

int main(void) {
    float myNumber = 1337.1;

    std::cout << myNumber << std::endl;

    return 0;
}
```

Let's review last week's tutorial.

```
pi@pi-alpha:~/code $ gdb -q example6
Reading symbols from example6...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example6
disas
Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    vldr    s0, [r11, #-8]
   0x0001070c <+28>:    bl      0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    strtmi  r2, [r7], #819  ; 0x333
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break on **main+20** and continue to that point.

```
(gdb) b *main+20
Breakpoint 2 at 0x10704
(gdb) c
Continuing.

Breakpoint 2, 0x00010704 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
=> 0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    vldr    s0, [r11, #-8]
   0x0001070c <+28>:    bl      0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    strtmi  r2, [r7], #819  ; 0x333
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's examine what value is inside **r11-8**. We clearly see it is **1337.09998** which approximates our value in our original c++ code. Keep in mind a float has roughly 7 decimal digits of precision and that is why we do not see **1337.1** so please remember that as we go forward.

```
(gdb) x/f $r11-8
0x7efff234:     1337.09998
```

We can also see this value in high memory.

```
(gdb) x/f 0x7efff234
0x7efff234:     1337.09998
```

Let's break on **main+28** and continue.

```
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

We see a strange new instruction. We see **vldr** and the value within **r11, #8** being moved into **s0**. So what is **s0**? We have a math co-processor which has a series of additional registers that work with decimal or floating-point numbers. Here we see an example of such to which the value of **1337.09998** is being moved into **s0**. The **vldr** instruction loads a constant value into every element of a single-precision or double-precision register such as **s0**.

```
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r3, [pc, #44]   ; 0x10730 <main+64>
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r0, [pc, #40]   ; 0x10734 <main+68>
   0x00010708 <+24>:    vldr    s0, [r11, #-8]
=> 0x0001070c <+28>:    bl      0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010710 <+32>:    mov     r3, r0
   0x00010714 <+36>:    mov     r0, r3
   0x00010718 <+40>:    ldr     r1, [pc, #24]   ; 0x10738 <main+72>
   0x0001071c <+44>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
   0x00010720 <+48>:    mov     r3, #0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    sub     sp, r11, #4
   0x0001072c <+60>:    pop     {r11, pc}
   0x00010730 <+64>:    strtmi  r2, [r7], #819  ; 0x333
   0x00010734 <+68>:    andeq   r0, r2, r0, lsr #19
   0x00010738 <+72>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We can only see these special registers if we do a info registers all command as we do below.

```
(gdb) i r a
```

Below we see the value now being moved into **s0**.

```
s0             1337.09998       (raw 0x44a72333)
s1             0        (raw 0x00000000)
s2             0        (raw 0x00000000)
s3             0        (raw 0x00000000)
s4             0        (raw 0x00000000)
s5             0        (raw 0x00000000)
s6             0        (raw 0x00000000)
s7             0        (raw 0x00000000)
s8             0        (raw 0x00000000)
s9             0        (raw 0x00000000)
s10            0        (raw 0x00000000)
s11            0        (raw 0x00000000)
s12            0        (raw 0x00000000)
s13            2        (raw 0x40000000)
s14            0        (raw 0x00000000)
s15            0        (raw 0x00000000)
s16            0        (raw 0x00000000)
s17            0        (raw 0x00000000)
s18            0        (raw 0x00000000)
s19            0        (raw 0x00000000)
s20            0        (raw 0x00000000)
s21            0        (raw 0x00000000)
s22            0        (raw 0x00000000)
s23            0        (raw 0x00000000)
s24            0        (raw 0x00000000)
s25            0        (raw 0x00000000)
s26            0        (raw 0x00000000)
s27            0        (raw 0x00000000)
s28            0        (raw 0x00000000)
s29            0        (raw 0x00000000)
s30            0        (raw 0x00000000)
s31            0        (raw 0x00000000)
```

Let's hack!

```
(gdb) set $s0 = 666.666
```

Let's now look at the registers and see what has transpired.

```
(gdb) i r a
```

```
s0             666.665955       (raw 0x4426aa9f)
s1             0        (raw 0x00000000)
s2             0        (raw 0x00000000)
s3             0        (raw 0x00000000)
s4             0        (raw 0x00000000)
s5             0        (raw 0x00000000)
s6             0        (raw 0x00000000)
s7             0        (raw 0x00000000)
s8             0        (raw 0x00000000)
s9             0        (raw 0x00000000)
s10            0        (raw 0x00000000)
s11            0        (raw 0x00000000)
s12            0        (raw 0x00000000)
s13            2        (raw 0x40000000)
s14            0        (raw 0x00000000)
s15            0        (raw 0x00000000)
s16            0        (raw 0x00000000)
s17            0        (raw 0x00000000)
s18            0        (raw 0x00000000)
s19            0        (raw 0x00000000)
s20            0        (raw 0x00000000)
s21            0        (raw 0x00000000)
s22            0        (raw 0x00000000)
s23            0        (raw 0x00000000)
s24            0        (raw 0x00000000)
s25            0        (raw 0x00000000)
s26            0        (raw 0x00000000)
s27            0        (raw 0x00000000)
s28            0        (raw 0x00000000)
s29            0        (raw 0x00000000)
s30            0        (raw 0x00000000)
s31            0        (raw 0x00000000)
```

As you can see we have hacked the value (less the precision issue of the float variable accurate up to 6 decimal places)!

```
(gdb) c
Continuing.
666.666
[Inferior 1 (process 1508) exited normally]
```

Finally as we continue we see our hacked value echoed back out to the terminal when the c++ **cout** function executes.

Next week we will dive into Double Variables.

Part 32 – Double Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The next stage in our journey is that of double-precision floating-point variables.

A double-precision floating-point variable is different from a floating-point variable as it is 64-bits wide and 15-17 significant digits of precision.

Let's examine our code.

```
#include <iostream>



int main(void) {

            double myNumber = 1337.77;

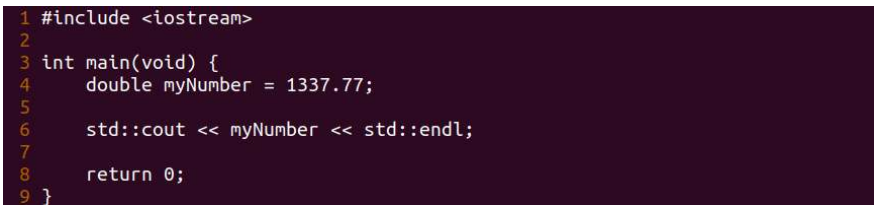


            std::cout << myNumber << std::endl;



            return 0;

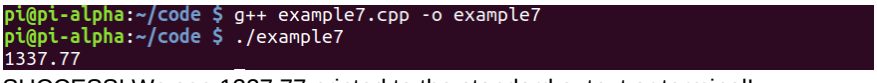
}
```



To compile this we simply type:

g++ example7.cpp -o example7

./example7



SUCCESS! We see 1337.77 printed to the standard output or terminal!

Let's break it down:

We assign the floating-point variable directly into the variable **myNumber** and then print it out to the terminal with the c++ **cout** function.

Next week we will dive into Debugging Double Variables.

Part 33 – Debugging Double Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code.

```
int main(void) {

            double myNumber = 1337.77;



            std::cout << myNumber << std::endl;



            return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     double myNumber = 1337.77;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

Let's debug!

```
pi@pi-alpha:~/code $ gdb -q example7
Reading symbols from example7...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example7

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r2, [pc, #48]   ; 0x10734 <main+68>
   0x00010700 <+16>:    ldr     r3, [pc, #48]   ; 0x10738 <main+72>
   0x00010704 <+20>:    strd    r2, [r11, #-12]
   0x00010708 <+24>:    ldr     r0, [pc, #44]   ; 0x1073c <main+76>
   0x0001070c <+28>:    vldr    d0, [r11, #-12]
   0x00010710 <+32>:    bl      0x1055c
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #28]   ; 0x10740 <main+80>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    bvc     0xff8625f4
   0x00010738 <+72>:    addsmi  lr, r4, r4, lsl r7
   0x0001073c <+76>:    andeq   r0, r2, r8, lsr #19
   0x00010740 <+80>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's set a breakpoint at **main+24** and continue.

```
(gdb) b *main+24
Breakpoint 2 at 0x10708
(gdb) c
Continuing.

Breakpoint 2, 0x00010708 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r2, [pc, #48]   ; 0x10734 <main+68>
   0x00010700 <+16>:    ldr     r3, [pc, #48]   ; 0x10738 <main+72>
   0x00010704 <+20>:    strd    r2, [r11, #-12]
=> 0x00010708 <+24>:    ldr     r0, [pc, #44]   ; 0x1073c <main+76>
   0x0001070c <+28>:    vldr    d0, [r11, #-12]
   0x00010710 <+32>:    bl      0x1055c
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #28]   ; 0x10740 <main+80>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    bvc     0xff8625f4
   0x00010738 <+72>:    addsmi  lr, r4, r4, lsl r7
   0x0001073c <+76>:    andeq   r0, r2, r8, lsr #19
   0x00010740 <+80>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We see the **strd r2, [r11, #-12]** and we have to fully understand that this means we are storing the value at the offset of **-12** from register **r11** into **r2**. Let's now examine what exactly resides there.

```
(gdb) x/s $r11-12
0x7efff230:     "\256G\341z\024\347\224@"
(gdb) x/f $r11-12
0x7efff230:     1337.77
```

Voila! We see **1337.77** at that offset location or specifically stored into **0x7efff230** in memory.

```
(gdb) si
0x0001070c in main ()
(gdb) si
0x00010710 in main ()
```

Let's step into twice which executes the **vldr d0, [r11, #-12]** as we understand that **1337.77** will now be loaded into the double precision math co-processor **d0** register. Let's now print the value at that location below.

```
(gdb) p $d0
$1 = {u8 = {174, 71, 225, 122, 20, 231, 148, 64}, u16 = {18350, 31457, 59156,
    16532}, u32 = {2061584302, 1083500308}, u64 = 4653598390127511470, f32 = {
    5.84860315e+35, 4.65320778}, f64 = 1337.77}
```

Finally let's continue and watch the value echo to the terminal. This completes our **cout** c++ function.

```
Breakpoint 2, 0x00010708 in main ()
(gdb) c
Continuing.
1337.77
[Inferior 1 (process 1199) exited normally]
```

Next week we will dive into Hacking Double Variables.

Part 34 – Hacking Double Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code.

```
int main(void) {

            double myNumber = 1337.77;



            std::cout << myNumber << std::endl;



            return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     double myNumber = 1337.77;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

Let's debug!

```
pi@pi-alpha:~/code $ gdb -q example7
Reading symbols from example7...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example7

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r2, [pc, #48]   ; 0x10734 <main+68>
   0x00010700 <+16>:    ldr     r3, [pc, #48]   ; 0x10738 <main+72>
   0x00010704 <+20>:    strd    r2, [r11, #-12]
   0x00010708 <+24>:    ldr     r0, [pc, #44]   ; 0x1073c <main+76>
   0x0001070c <+28>:    vldr    d0, [r11, #-12]
   0x00010710 <+32>:    bl      0x1055c
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #28]   ; 0x10740 <main+80>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    bvc     0xff8625f4
   0x00010738 <+72>:    addsmi  lr, r4, r4, lsl r7
   0x0001073c <+76>:    andeq   r0, r2, r8, lsr #19
   0x00010740 <+80>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's set a breakpoint at **main+24** and continue.

```
(gdb) b *main+24
Breakpoint 2 at 0x10708
(gdb) c
Continuing.

Breakpoint 2, 0x00010708 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    ldr     r2, [pc, #48]   ; 0x10734 <main+68>
   0x00010700 <+16>:    ldr     r3, [pc, #48]   ; 0x10738 <main+72>
   0x00010704 <+20>:    strd    r2, [r11, #-12]
=> 0x00010708 <+24>:    ldr     r0, [pc, #44]   ; 0x1073c <main+76>
   0x0001070c <+28>:    vldr    d0, [r11, #-12]
   0x00010710 <+32>:    bl      0x1055c
   0x00010714 <+36>:    mov     r3, r0
   0x00010718 <+40>:    mov     r0, r3
   0x0001071c <+44>:    ldr     r1, [pc, #28]   ; 0x10740 <main+80>
   0x00010720 <+48>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010724 <+52>:    mov     r3, #0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    sub     sp, r11, #4
   0x00010730 <+64>:    pop     {r11, pc}
   0x00010734 <+68>:    bvc     0xff8625f4
   0x00010738 <+72>:    addsmi  lr, r4, r4, lsl r7
   0x0001073c <+76>:    andeq   r0, r2, r8, lsr #19
   0x00010740 <+80>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We see the **strd r2, [r11, #-12]** and we have to fully understand that this means we are storing the value at the offset of **-12** from register **r11** into **r2**. Let's now examine what exactly resides there.

```
(gdb) x/s $r11-12
0x7efff230:     "\256G\341z\024\347\224@"
(gdb) x/f $r11-12
0x7efff230:     1337.77
```

Voila! We see **1337.77** at that offset location or specifically stored into **0x7efff230** in memory.

```
(gdb) si
0x0001070c in main ()
(gdb) si
0x00010710 in main ()
```

Let's step into twice which executes the **vldr d0, [r11, #-12]** as we understand that **1337.77** will now be loaded into the double precision math coprocessor **d0** register. Let's now print the value at that location below.

```
(gdb) p $d0
$1 = {u8 = {174, 71, 225, 122, 20, 231, 148, 64}, u16 = {18350, 31457, 59156,
    16532}, u32 = {2061584302, 1083500308}, u64 = 4653598390127511470, f32 = {
    5.84860315e+35, 4.65320778}, f64 = 1337.77}
```

Let's hack the **d0** register!

```
(gdb) set $d0 = 666.66
```

Now let's reexamine the value inside **d0**.

```
(gdb) p $d0
$2 = {u8 = {225, 122, 20, 174, 71, 213, 132, 64}, u16 = {31457, 44564, 54599,
    16516}, u32 = {2920577761, 1082447175}, u64 = 4649075219193166561, f32 = {
    -3.37604424e-11, 4.15103483}, f64 = 666.65999999999997}
```

Let's continue.

```
(gdb) c
Continuing.
666.66
[Inferior 1 (process 1140) exited normally]
```

Successfully hacked!

Next week we will dive into the SizeOf Operator.

Part 35 – SizeOf Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The next stage in our journey is that of the SizeOf operator.

Let's examine our code.

```
#include <iostream>



int main(void) {

            int myNumber = 16;

            int myNumberSize = sizeof(myNumber);

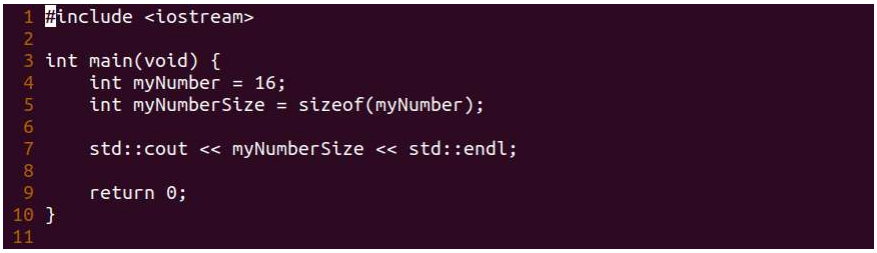


            std::cout << myNumberSize << std::endl;



            return 0;

}
```



```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNumberSize = sizeof(myNumber);

    std::cout << myNumberSize << std::endl;

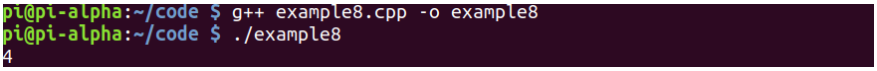
    return 0;
}

```

To compile this we simply type:

g++ example8.cpp -o example8

./example8



```
pi@pi-alpha:~/code $ g++ example8.cpp -o example8
pi@pi-alpha:~/code $ ./example8
4
```

We see 4 printed to the screen.

Let's break it down:

We create a variable **myNumber = 16** to which we create another variable **myNumberSize** which holds the value of the size of **myNumber**. We see that when we execute our code it shows 4 therefore we see that the SizeOf operator indicates an integer is 4 bytes wide.

Next week we will dive into Debugging SizeOf Operator.

Part 36 – Debugging SizeOf Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>



int main(void) {

            int myNumber = 16;

            int myNumberSize = sizeof(myNumber);

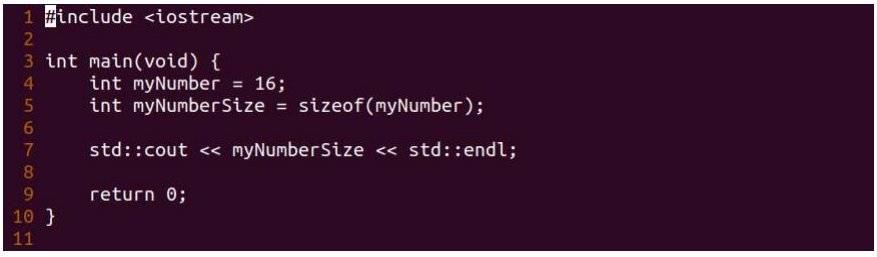


            std::cout << myNumberSize << std::endl;



            return 0;

}
```



```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNumberSize = sizeof(myNumber);

    std::cout << myNumberSize << std::endl;

    return 0;
}

```

Remember that we create a variable **myNumber = 16** to which we create another variable **myNumberSize** which holds the value of the size of **myNumber**. We see that when we execute our code it shows 4 therefore we see that the SizeOf operator indicates an integer is 4 bytes wide.

Let's debug and break on main.

```
pi@pi-alpha:~/code $ gdb -q example8
Reading symbols from example8...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example8

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    mov     r3, #4
   0x00010708 <+24>:    str     r3, [r11, #-12]
   0x0001070c <+28>:    ldr     r0, [pc, #36]   ; 0x10738 <main+72>
   0x00010710 <+32>:    ldr     r1, [r11, #-12]
   0x00010714 <+36>:    bl      0x1055c
   0x00010718 <+40>:    mov     r3, r0
   0x0001071c <+44>:    mov     r0, r3
   0x00010720 <+48>:    ldr     r1, [pc, #20]   ; 0x1073c <main+76>
   0x00010724 <+52>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010728 <+56>:    mov     r3, #0
   0x0001072c <+60>:    mov     r0, r3
   0x00010730 <+64>:    sub     sp, r11, #4
   0x00010734 <+68>:    pop     {r11, pc}
   0x00010738 <+72>:    andeq   r0, r2, r8, lsr #19
   0x0001073c <+76>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break on **main+20** as we can see the value of **4** being moved into **r3**.

```
(gdb) b *main+20
Breakpoint 2 at 0x10704
(gdb) c
Continuing.

Breakpoint 2, 0x00010704 in main ()
(gdb) x/d $r11-8
0x7efff234:     16
```

Let's examine what is going on at **main+16** as we can see that we are storing into the value of **$r11-8** that which exists in **r3** which in our case is **16**. This makes sense as when we examine our original code the value of **myNumber** was in fact **16**. We can see this here when we examine the value inside **$r11-8**.

```
(gdb) si
0x00010708 in main ()
(gdb) si
0x0001070c in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    mov     r3, #4
   0x00010708 <+24>:    str     r3, [r11, #-12]
=> 0x0001070c <+28>:    ldr     r0, [pc, #36]   ; 0x10738 <main+72>
   0x00010710 <+32>:    ldr     r1, [r11, #-12]
   0x00010714 <+36>:    bl      0x1055c
   0x00010718 <+40>:    mov     r3, r0
   0x0001071c <+44>:    mov     r0, r3
   0x00010720 <+48>:    ldr     r1, [pc, #20]   ; 0x1073c <main+76>
   0x00010724 <+52>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010728 <+56>:    mov     r3, #0
   0x0001072c <+60>:    mov     r0, r3
   0x00010730 <+64>:    sub     sp, r11, #4
   0x00010734 <+68>:    pop     {r11, pc}
   0x00010738 <+72>:    andeq   r0, r2, r8, lsr #19
   0x0001073c <+76>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) x/d $r11-12
0x7efff230:     4
```

As we can see above the value inside **$r11-12** is **4** as that represents the value that **SizeOf** is returning as the integer **16** is in fact 4 bytes wide.

```
(gdb) x/d $r11-12
0x7efff230:     4
(gdb) c
Continuing.
4
[Inferior 1 (process 1114) exited normally]
```

Finally when we continue execution we in fact see the value **4** echoed to the terminal.

Next week we will dive into Hacking SizeOf Operator.

Part 37 – Hacking SizeOf Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>



int main(void) {

            int myNumber = 16;

            int myNumberSize = sizeof(myNumber);

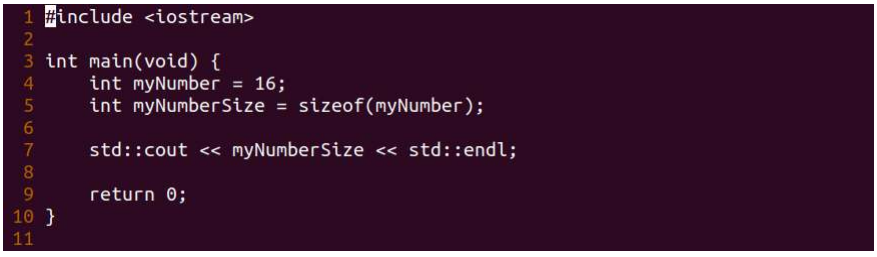


            std::cout << myNumberSize << std::endl;



            return 0;

}
```



```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNumberSize = sizeof(myNumber);

    std::cout << myNumberSize << std::endl;

    return 0;
}

```

Remember that we create a variable **myNumber = 16** to which we create another variable **myNumberSize** which holds the value of the size of **myNumber**. We see that when we execute our code it shows 4 therefore we see that the SizeOf operator indicates an integer is 4 bytes wide.

Let's review last week's code as we start with debugging and breaking on main.

```
pi@pi-alpha:~/code $ gdb -q example8
Reading symbols from example8...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example8

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    mov     r3, #4
   0x00010708 <+24>:    str     r3, [r11, #-12]
   0x0001070c <+28>:    ldr     r0, [pc, #36]   ; 0x10738 <main+72>
   0x00010710 <+32>:    ldr     r1, [r11, #-12]
   0x00010714 <+36>:    bl      0x1055c
   0x00010718 <+40>:    mov     r3, r0
   0x0001071c <+44>:    mov     r0, r3
   0x00010720 <+48>:    ldr     r1, [pc, #20]   ; 0x1073c <main+76>
   0x00010724 <+52>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010728 <+56>:    mov     r3, #0
   0x0001072c <+60>:    mov     r0, r3
   0x00010730 <+64>:    sub     sp, r11, #4
   0x00010734 <+68>:    pop     {r11, pc}
   0x00010738 <+72>:    andeq   r0, r2, r8, lsr #19
   0x0001073c <+76>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break on **main+20** as we can see the value of **4** being moved into **r3**.

```
(gdb) b *main+20
Breakpoint 2 at 0x10704
(gdb) c
Continuing.

Breakpoint 2, 0x00010704 in main ()
(gdb) x/d $r11-8
0x7efff234:     16
```

Let's examine what is going on at **main+16** as we can see that we are storing into the value of **$r11-8** that which exists in **r3** which in our case is **16**. This makes sense as when we examine our original code the value of **myNumber** was in fact **16**. We can see this here when we examine the value inside **$r11-8**.

```
(gdb) si
0x00010710 in main ()
(gdb) si
0x00010714 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    mov     r3, #4
   0x00010708 <+24>:    str     r3, [r11, #-12]
   0x0001070c <+28>:    ldr     r0, [pc, #36]   ; 0x10738 <main+72>
   0x00010710 <+32>:    ldr     r1, [r11, #-12]
=> 0x00010714 <+36>:    bl      0x1055c
   0x00010718 <+40>:    mov     r3, r0
   0x0001071c <+44>:    mov     r0, r3
   0x00010720 <+48>:    ldr     r1, [pc, #20]   ; 0x1073c <main+76>
   0x00010724 <+52>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010728 <+56>:    mov     r3, #0
   0x0001072c <+60>:    mov     r0, r3
   0x00010730 <+64>:    sub     sp, r11, #4
   0x00010734 <+68>:    pop     {r11, pc}
   0x00010738 <+72>:    andeq   r0, r2, r8, lsr #19
   0x0001073c <+76>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

As we can see above the value inside **$r11-12** is **4** as that represents the value that **SizeOf** is returning as the integer **16** is in fact 4 bytes wide.

```
(gdb) x/d $r11-12
0x7efff230:     4
(gdb) c
Continuing.
4
[Inferior 1 (process 1114) exited normally]
```

Finally when we continue execution we in fact see the value **4** echoed to the terminal.

Let's hack!

```
pi@pi-alpha:~/code $ gdb -q example8
Reading symbols from example8...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example8

Breakpoint 1, 0x000106f0 in main ()
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    mov     r3, #4
   0x00010708 <+24>:    str     r3, [r11, #-12]
=> 0x0001070c <+28>:    ldr     r0, [pc, #36]   ; 0x10738 <main+72>
   0x00010710 <+32>:    ldr     r1, [r11, #-12]
   0x00010714 <+36>:    bl      0x1055c
   0x00010718 <+40>:    mov     r3, r0
   0x0001071c <+44>:    mov     r0, r3
   0x00010720 <+48>:    ldr     r1, [pc, #20]   ; 0x1073c <main+76>
   0x00010724 <+52>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010728 <+56>:    mov     r3, #0
   0x0001072c <+60>:    mov     r0, r3
   0x00010730 <+64>:    sub     sp, r11, #4
   0x00010734 <+68>:    pop     {r11, pc}
   0x00010738 <+72>:    andeq   r0, r2, r8, lsr #19
   0x0001073c <+76>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We run and break on **main+28**.

```
(gdb) print $r3
$1 = 4
```

We see the value in **r3** is **4** which is expected.

```
(gdb) si
0x00010710 in main ()
(gdb) si
0x00010714 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    mov     r3, #4
   0x00010708 <+24>:    str     r3, [r11, #-12]
   0x0001070c <+28>:    ldr     r0, [pc, #36]   ; 0x10738 <main+72>
   0x00010710 <+32>:    ldr     r1, [r11, #-12]
=> 0x00010714 <+36>:    bl      0x1055c
   0x00010718 <+40>:    mov     r3, r0
   0x0001071c <+44>:    mov     r0, r3
   0x00010720 <+48>:    ldr     r1, [pc, #20]   ; 0x1073c <main+76>
   0x00010724 <+52>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010728 <+56>:    mov     r3, #0
   0x0001072c <+60>:    mov     r0, r3
   0x00010730 <+64>:    sub     sp, r11, #4
   0x00010734 <+68>:    pop     {r11, pc}
   0x00010738 <+72>:    andeq   r0, r2, r8, lsr #19
   0x0001073c <+76>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We break on **main+36**.

```
(gdb) print $r1
$5 = 4
```

We see the value in **r1** is **4** which should make logical sense as the value was stored from **r3** into **r11-12** and then back to **r1**.

```
(gdb) set $r1 = 666
```

Let's hack the value in **r1**!

```
(gdb) c
Continuing.
666
[Inferior 1 (process 1149) exited normally]
```

Success! We have hacked the machine!

Next week we will dive into the Pre-Increment Operator.

Part 38 – Pre-Increment Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The next stage in our journey is that of the pre-increment operator.

Let's examine our code.

```
#include <iostream>



int main(void) {

            int myNumber = 16;

            int myNewNumber = ++myNumber;

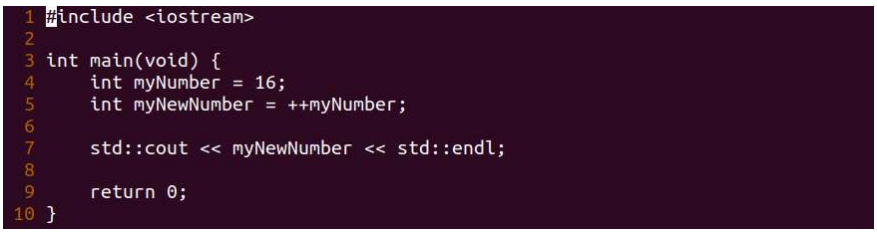


            std::cout << myNewNumber << std::endl;



            return 0;

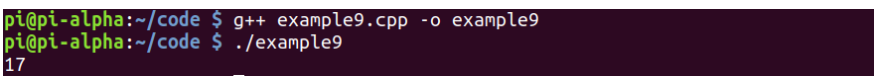
}
```



To compile this we simply type:

g++ example9.cpp -o example9

./example9



We see 17 printed to the screen.

Let's break it down:

We create a variable **myNumber = 16** to which we create another variable **myNewNumber** which pre-increments the value of **myNumber**. We see that when we execute our code it shows 17.

When we pre-increment the value of the variable is incremented before assigning it to another variable. For example **myNumber** is **16** so it gets incremented before being assigned to **myNewNumber** so therefore we get **17**.

Next week we will dive into Debugging Pre-Increment Operator.

Part 39 – Debugging Pre-Increment Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>



int main(void) {

            int myNumber = 16;

            int myNewNumber = ++myNumber;

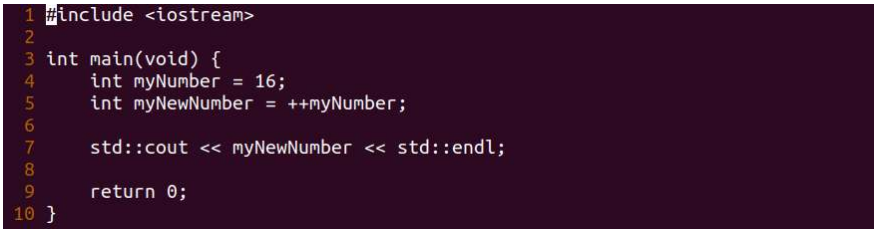


            std::cout << myNewNumber << std::endl;



            return 0;

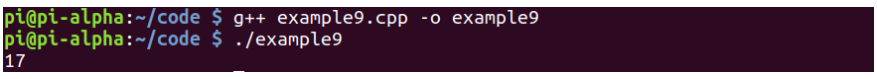
}
```



To compile this we simply type:

g++ example9.cpp -o example9

./example9



We see 17 printed to the screen.

Let's break it down:

We create a variable **myNumber = 16** to which we create another variable **myNewNumber** which pre-increments the value of **myNumber**. We see that when we execute our code it shows 17.

When we pre-increment the value of the variable is incremented before assigning it to another variable. For example **myNumber** is **16** so it gets incremented before being assigned to **myNewNumber** so therefore we get **17**.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q example9
Reading symbols from example9...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example9

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    add     r3, r3, #1
   0x0001070c <+28>:    str     r3, [r11, #-8]
   0x00010710 <+32>:    ldr     r3, [r11, #-8]
   0x00010714 <+36>:    str     r3, [r11, #-12]
   0x00010718 <+40>:    ldr     r0, [pc, #36]   ; 0x10744 <main+84>
   0x0001071c <+44>:    ldr     r1, [r11, #-12]
   0x00010720 <+48>:    bl      0x1055c
   0x00010724 <+52>:    mov     r3, r0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    ldr     r1, [pc, #20]   ; 0x10748 <main+88>
   0x00010730 <+64>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010734 <+68>:    mov     r3, #0
   0x00010738 <+72>:    mov     r0, r3
   0x0001073c <+76>:    sub     sp, r11, #4
   0x00010740 <+80>:    pop     {r11, pc}
   0x00010744 <+84>:                    ; <UNDEFINED> instruction: 0x000209b0
   0x00010748 <+88>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We do our normal start in gdb and break on main. Take note at **main+24** we are moving the value of **1** into **r3**. We then see at **main+28** we are storing that value at **r11-8** to which we will set a breakpoint and continue.

```
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

As we evaluate the value in **r3** at this stage we see **17**. Remember back in our original code that the value in the **myNumber** variable was **16**. We can see that the pre-increment operator was successful to increment the value **1** to give us **17**.

```
(gdb) print $r3
$1 = 17
```

We see that when we continue through the code the value **17** is successfully echoed to the terminal as expected.

```
(gdb) c
Continuing.
17
[Inferior 1 (process 1027) exited normally]
```

Next week we will dive into Hacking Debugging Pre-Increment Operator.

Part 40 – Hacking Pre-Increment Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's one again re-examine our code.

```
#include <iostream>



int main(void) {

            int myNumber
= 16;

            int
myNewNumber = ++myNumber;

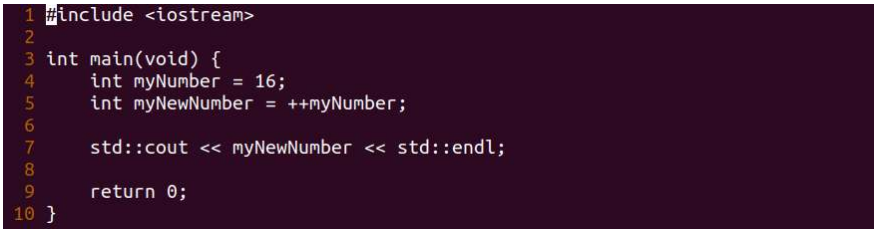


            std::cout
<< myNewNumber << std::endl;



            return 0;

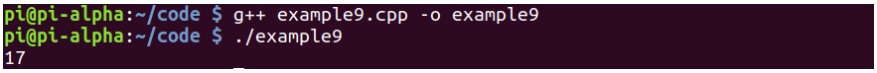
}
```



To compile this we simply type:

g++ example9.cpp -o example9

./example9



We see 17 printed to the screen.

Let's break it down:

We create a variable **myNumber = 16** to which we create another variable **myNewNumber** which pre-increments the value of **myNumber**. We see that when we execute our code it shows 17.

When we pre-increment the value of the variable is incremented before assigning it to another variable. For example **myNumber** is **16** so it gets incremented before being assigned to **myNewNumber** so therefore we get **17**.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q example9
Reading symbols from example9...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example9

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    add     r3, r3, #1
   0x0001070c <+28>:    str     r3, [r11, #-8]
   0x00010710 <+32>:    ldr     r3, [r11, #-8]
   0x00010714 <+36>:    str     r3, [r11, #-12]
   0x00010718 <+40>:    ldr     r0, [pc, #36]   ; 0x10744 <main+84>
   0x0001071c <+44>:    ldr     r1, [r11, #-12]
   0x00010720 <+48>:    bl      0x1055c
   0x00010724 <+52>:    mov     r3, r0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    ldr     r1, [pc, #20]   ; 0x10748 <main+88>
   0x00010730 <+64>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010734 <+68>:    mov     r3, #0
   0x00010738 <+72>:    mov     r0, r3
   0x0001073c <+76>:    sub     sp, r11, #4
   0x00010740 <+80>:    pop     {r11, pc}
   0x00010744 <+84>:                    ; <UNDEFINED> instruction: 0x000209b0
   0x00010748 <+88>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We do our normal start in gdb and break on main. Take note at **main+24** we are moving the value of **1** into **r3**. We then see at **main+28** we are storing that value at **r11-8** to which we will set a breakpoint and continue.

```
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

As we evaluate the value in **r3** at this stage we see **17**. Remember back in our original code that the value in the **myNumber** variable was **16**. We can see that the pre-increment operator was successful to increment the value **1** to give us **17**.

```
(gdb) print $r3
$1 = 17
```

We see that when we continue through the code the value **17** is successfully echoed to the terminal as expected.

```
(gdb) c
Continuing.
17
[Inferior 1 (process 1027) exited normally]
```

Let's re-run the program.

```
(gdb) r
Starting program: /home/pi/code/example9

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    add     r3, r3, #1
   0x0001070c <+28>:    str     r3, [r11, #-8]
   0x00010710 <+32>:    ldr     r3, [r11, #-8]
   0x00010714 <+36>:    str     r3, [r11, #-12]
   0x00010718 <+40>:    ldr     r0, [pc, #36]   ; 0x10744 <main+84>
   0x0001071c <+44>:    ldr     r1, [r11, #-12]
   0x00010720 <+48>:    bl      0x1055c
   0x00010724 <+52>:    mov     r3, r0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    ldr     r1, [pc, #20]   ; 0x10748 <main+88>
   0x00010730 <+64>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010734 <+68>:    mov     r3, #0
   0x00010738 <+72>:    mov     r0, r3
   0x0001073c <+76>:    sub     sp, r11, #4
   0x00010740 <+80>:    pop     {r11, pc}
   0x00010744 <+84>:                    ; <UNDEFINED> instruction: 0x000209b0
   0x00010748 <+88>:                    ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's hack! Here were review the value in r3 which we know to be 17. Let's hack it to something else.

```
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
(gdb) print $r3
$2 = 17
(gdb) set $r3 = 666
```

Success! As we can see when we continue we now see the hacked value echoing to the terminal.

```
(gdb) c
Continuing.
666
[Inferior 1 (process 1051) exited normally]
```

Next week we will dive into the Post-Increment Operator.

Part 41 – Post-Increment Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's dive into our code.

```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNewNumber = ++myNumber;

    std::cout << myNewNumber << std::endl;

    return 0;
}
```

```
1  #include <iostream>
2
3  int main(void) {
4      int myNumber = 16;
5      int myNewNumber = myNumber++;
6
7      std::cout << myNewNumber << std::endl;
8      std::cout << myNumber << std::endl;
9
10     return 0;
11 }
```

To compile this we simply type:

g++ example10.cpp -o example10

./example10

```
pi@pi-alpha:~/code $ g++ example10.cpp -o example10
pi@pi-alpha:~/code $ ./example10
16
17
```

We see 16 and 17 printed to the screen.

Let's break it down:

We create a variable **myNumber = 16** to which we create another variable **myNewNumber** which post-increments the value of **myNumber**. We see that when we execute our code it shows **16** as the value of **myNewNumber** and **17** as the value of **myNumber** as **myNewNumber** does not get incremented as only **myNumber** get incremented as it is a post operator.

When we post-increment the value of the variable is incremented after assigning it to another variable. For example **myNumber** is **16** so it gets incremented after being assigned to **myNewNumber** so therefore we get **17**.

Next week we will dive into Debugging Post-Increment Operator.

Part 42 – Debugging Post-Increment Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNewNumber = ++myNumber;

    std::cout << myNewNumber << std::endl;

    return 0;
}
```

We create a variable **myNumber = 16** to which we create another variable **myNewNumber** which post-increments the value of **myNumber**. We see that when we execute our code it shows **16** as the value of **myNewNumber** and **17** as the value of **myNumber** as **myNewNumber** does not get incremented as only **myNumber** get incremented as it is a post operator.

When we post-increment the value of the variable is incremented after assigning it to another variable. For example **myNumber** is **16** so it gets incremented after being assigned to **myNewNumber** so therefore we get **17**.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q example10
Reading symbols from example10...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example10

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    add     r2, r3, #1
   0x0001070c <+28>:    str     r2, [r11, #-8]
   0x00010710 <+32>:    str     r3, [r11, #-12]
   0x00010714 <+36>:    ldr     r0, [pc, #64]   ; 0x1075c <main+108>
   0x00010718 <+40>:    ldr     r1, [r11, #-12]
   0x0001071c <+44>:    bl      0x1055c
   0x00010720 <+48>:    mov     r3, r0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    ldr     r1, [pc, #48]   ; 0x10760 <main+112>
   0x0001072c <+60>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010730 <+64>:    ldr     r0, [pc, #36]   ; 0x1075c <main+108>
   0x00010734 <+68>:    ldr     r1, [r11, #-8]
   0x00010738 <+72>:    bl      0x1055c
   0x0001073c <+76>:    mov     r3, r0
   0x00010740 <+80>:    mov     r0, r3
   0x00010744 <+84>:    ldr     r1, [pc, #20]   ; 0x10760 <main+112>
   0x00010748 <+88>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x0001074c <+92>:    mov     r3, #0
   0x00010750 <+96>:    mov     r0, r3
   0x00010754 <+100>:   sub     sp, r11, #4
   0x00010758 <+104>:   pop     {r11, pc}
   0x0001075c <+108>:   andeq   r0, r2, r8, asr #19
   0x00010760 <+112>:                   ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break on ***main+28** and continue.

```
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

As we can see the value in **r3** is 16 and the value in **r2** is 17. We can see that as they are loaded from memory into the registers in ***main+12** directly by the **mov** instruction and ***main+24** we add 1 into **r3** and then put that value into **r2**.

```
(gdb) print $r3
$1 = 16
(gdb) print $r2
$2 = 17
```

As we continue we can see the **cout** c++ function called which echos out the values to the terminal (standard output) as expected.

```
(gdb) c
Continuing.
16
17
[Inferior 1 (process 1018) exited normally]
```

Next week we will dive into Hacking Post-Increment Operator.

Part 43 – Hacking Post-Increment Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNewNumber = ++myNumber;

    std::cout << myNewNumber << std::endl;

    return 0;
}
```

We create a variable **myNumber = 16** to which we create another variable **myNewNumber** which post-increments the value of **myNumber**. We see that when we execute our code it shows **16** as the value of **myNewNumber** and **17** as the value of **myNumber** as **myNewNumber** does not get incremented as only **myNumber** get incremented as it is a post operator.

When we post-increment the value of the variable is incremented after assigning it to another variable. For example **myNumber** is **16** so it gets incremented after being assigned to **myNewNumber** so therefore we get **17**.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q example10
Reading symbols from example10...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example10

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    add     r2, r3, #1
   0x0001070c <+28>:    str     r2, [r11, #-8]
   0x00010710 <+32>:    str     r3, [r11, #-12]
   0x00010714 <+36>:    ldr     r0, [pc, #64]   ; 0x1075c <main+108>
   0x00010718 <+40>:    ldr     r1, [r11, #-12]
   0x0001071c <+44>:    bl      0x1055c
   0x00010720 <+48>:    mov     r3, r0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    ldr     r1, [pc, #48]   ; 0x10760 <main+112>
   0x0001072c <+60>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010730 <+64>:    ldr     r0, [pc, #36]   ; 0x1075c <main+108>
   0x00010734 <+68>:    ldr     r1, [r11, #-8]
   0x00010738 <+72>:    bl      0x1055c
   0x0001073c <+76>:    mov     r3, r0
   0x00010740 <+80>:    mov     r0, r3
   0x00010744 <+84>:    ldr     r1, [pc, #20]   ; 0x10760 <main+112>
   0x00010748 <+88>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x0001074c <+92>:    mov     r3, #0
   0x00010750 <+96>:    mov     r0, r3
   0x00010754 <+100>:   sub     sp, r11, #4
   0x00010758 <+104>:   pop     {r11, pc}
   0x0001075c <+108>:   andeq   r0, r2, r8, asr #19
   0x00010760 <+112>:                   ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break on ***main+28** and continue.

```
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

As we can see the value in **r3** is **16** and the value in **r2** is **17**. We can see that as they are loaded from memory into the registers in ***main+12** directly by the **mov** instruction and ***main+24** we add **1** into **r3** and then put that value into **r2**.

Let's hack this baby!

```
(gdb) set $r3 = 666
(gdb) set $r2 = 666
(gdb) print $r3
$3 = 666
(gdb) print $r2
$4 = 666
```

We know we can now set the value of **r3** to our heart's desire!

```
(gdb) c
Continuing.
666
666
[Inferior 1 (process 1026) exited normally]
```

As we continue we see the c++ **cout** function echo our new hacked value to the screen!

Next week we will dive into the Pre-Decrement Operator.

Part 44 – Pre-Decrement Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's take a look at our pre-decrement operator example. The pre-decrement operator decrements a given value before the action gets assigned.

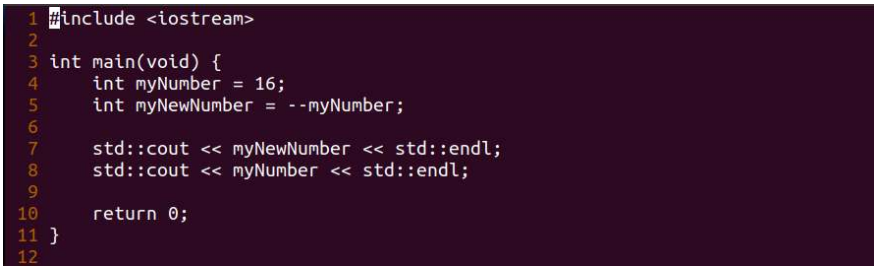
Let's examine our code.

```
#include <iostream>

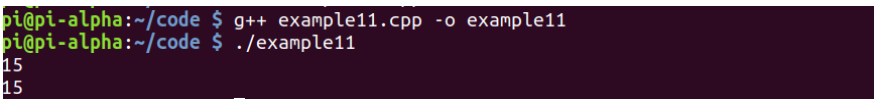
int main(void) {
    int myNumber = 16;
    int myNewNumber = --myNumber;

    std::cout << myNewNumber << std::endl;
    std::cout << myNumber << std::endl;

    return 0;
}
```



As we compile and run we see 15 echoed out to the terminal.



The value of **myNumber** was **16** and when it is assigned with the pre-decrement operator we see that the new value is **15** as it is assigned into **myNewNumber**.

Next week we will dive into the Debuggin Pre-Decrement Operator.

Part 45 – Debugging Pre-Decrement Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNewNumber = --myNumber;

    std::cout << myNewNumber << std::endl;
    std::cout << myNumber << std::endl;

    return 0;
}
```

We remember when we compile we get 15.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q ./example11
Reading symbols from ./example11...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example11

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    sub     r3, r3, #1
   0x0001070c <+28>:    str     r3, [r11, #-8]
   0x00010710 <+32>:    ldr     r3, [r11, #-8]
   0x00010714 <+36>:    str     r3, [r11, #-12]
   0x00010718 <+40>:    ldr     r0, [pc, #64]   ; 0x10760 <main+112>
   0x0001071c <+44>:    ldr     r1, [r11, #-12]
   0x00010720 <+48>:    bl      0x1055c
   0x00010724 <+52>:    mov     r3, r0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    ldr     r1, [pc, #48]   ; 0x10764 <main+116>
   0x00010730 <+64>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010734 <+68>:    ldr     r0, [pc, #36]   ; 0x10760 <main+112>
   0x00010738 <+72>:    ldr     r1, [r11, #-8]
   0x0001073c <+76>:    bl      0x1055c
   0x00010740 <+80>:    mov     r3, r0
   0x00010744 <+84>:    mov     r0, r3
   0x00010748 <+88>:    ldr     r1, [pc, #20]   ; 0x10764 <main+116>
   0x0001074c <+92>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010750 <+96>:    mov     r3, #0
   0x00010754 <+100>:   mov     r0, r3
   0x00010758 <+104>:   sub     sp, r11, #4
   0x0001075c <+108>:   pop     {r11, pc}
   0x00010760 <+112>:   ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010764 <+116>:                   ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break.

```
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
(gdb) print $r3
$1 = 15
```

As we can see **r3** holds 15. Keep in mind hacking this value may not be the final place it may be stored. Remember this for next week and re-examine the debug code above to see if you can figure it out.

```
(gdb) b *main+48
Breakpoint 2 at 0x10720
(gdb) c
Continuing.

Breakpoint 2, 0x00010720 in main ()
(gdb) print $r1
$1 = 15
```

As we can see **r1** holds 15 as well. Keep in mind the above statement.

```
(gdb) c
Continuing.

Breakpoint 2, 0x00010720 in main ()
(gdb) print $r1
$1 = 15
(gdb) c
Continuing.
15
15
[Inferior 1 (process 986) exited normally]
```

As we continue we see our **cout** function echoing 15 for both areas as expected.

Next week we will dive into the Hacking Pre-Decrement Operator.

Part 46 – Hacking Pre-Decrement Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNewNumber = --myNumber;

    std::cout << myNewNumber << std::endl;
    std::cout << myNumber << std::endl;

    return 0;
}
```

We remember when we compile we get 15.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q ./example11
Reading symbols from ./example11...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example11

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    sub     r3, r3, #1
   0x0001070c <+28>:    str     r3, [r11, #-8]
   0x00010710 <+32>:    ldr     r3, [r11, #-8]
   0x00010714 <+36>:    str     r3, [r11, #-12]
   0x00010718 <+40>:    ldr     r0, [pc, #64]   ; 0x10760 <main+112>
   0x0001071c <+44>:    ldr     r1, [r11, #-12]
   0x00010720 <+48>:    bl      0x1055c
   0x00010724 <+52>:    mov     r3, r0
   0x00010728 <+56>:    mov     r0, r3
   0x0001072c <+60>:    ldr     r1, [pc, #48]   ; 0x10764 <main+116>
   0x00010730 <+64>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010734 <+68>:    ldr     r0, [pc, #36]   ; 0x10760 <main+112>
   0x00010738 <+72>:    ldr     r1, [r11, #-8]
   0x0001073c <+76>:    bl      0x1055c
   0x00010740 <+80>:    mov     r3, r0
   0x00010744 <+84>:    mov     r0, r3
   0x00010748 <+88>:    ldr     r1, [pc, #20]   ; 0x10764 <main+116>
   0x0001074c <+92>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010750 <+96>:    mov     r3, #0
   0x00010754 <+100>:   mov     r0, r3
   0x00010758 <+104>:   sub     sp, r11, #4
   0x0001075c <+108>:   pop     {r11, pc}
   0x00010760 <+112>:   ldrdeq  r0, [r2], -r0   ; <UNPREDICTABLE>
   0x00010764 <+116>:                   ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's break.

```
(gdb) b *main+40
Breakpoint 2 at 0x10718
(gdb) c
Continuing.

Breakpoint 2, 0x00010718 in main ()
```

Let's review what is inside **r3** and hack it.

```
(gdb) print $r3
$1 = 15
(gdb) set $r3 = 666
(gdb) print $r3
$2 = 666
```

Now as we continue we see it did not successfully hack why is that?

```
(gdb) c
Continuing.
15
15
[Inferior 1 (process 995) exited normally]
```

We re-run the binary and break and see the value here at **r1** hold **15**.

```
(gdb) b *main+48
Breakpoint 2 at 0x10720
(gdb) c
Continuing.

Breakpoint 2, 0x00010720 in main ()
(gdb) print $r1
$1 = 15
```

When we continue we see 15 which we don't want.

```
(gdb) c
Continuing.
15
15
[Inferior 1 (process 995) exited normally]
```

Now we break again and print the value.

```
(gdb) b *main+48
Breakpoint 2 at 0x10720
(gdb) c
Continuing.

Breakpoint 2, 0x00010720 in main ()
(gdb) print $r1
$1 = 15
```

This time we set **r1** and we can see we have successfully hacked!

```
(gdb) set $r1 = 666
(gdb) c
Continuing.
666
15
[Inferior 1 (process 1003) exited normally]
```

This is your first experience with really breaking down the registers and seeing where things are stored and how it can affect outcome. Take time and run this yourself so you really have a firm handle on this.

Next week we will dive into the Post-Decrement Operator.

Part 47 – Post-Decrement Operator

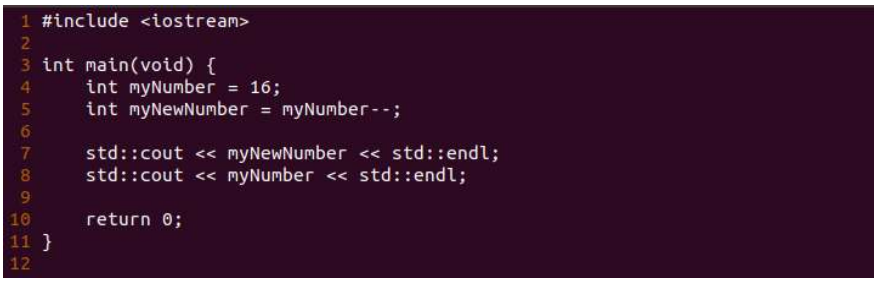
This week we will address the post-decrement operator. Let's examine our code.

```
#include <iostream>

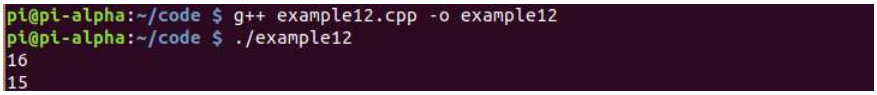
int main(void) {
    int myNumber = 16;
    int myNewNumber = myNumber--;

    std::cout << myNewNumber << std::endl;
    std::cout << myNumber << std::endl;

    return 0;
}
```



As we compile we see **16** and **15** printed out respectively.



We see that in this scenario **myNewNumber** does get decremented as **myNumber--** takes the value of 16 and reduces it to 15.

Next week we will dive into the Debugging Post-Decrement Operator.

Part 48 – Debugging Post-Decrement Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's re-examine our code.

```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNewNumber = myNumber--;

    std::cout << myNewNumber << std::endl;
    std::cout << myNumber << std::endl;

    return 0;
}
```

We see our very simple C++ code above to which we are doing nothing more than assigning a number into a variable to which we init another int variable and assign the original variable to which it is post-decremented. We then output each value to the terminal.

Let's debug.

```
pi@pi-alpha:~/code $ gdb -q example12
Reading symbols from example12...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example12

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    sub     r2, r3, #1
   0x0001070c <+28>:    str     r2, [r11, #-8]
   0x00010710 <+32>:    str     r3, [r11, #-12]
   0x00010714 <+36>:    ldr     r0, [pc, #64]   ; 0x1075c <main+108>
   0x00010718 <+40>:    ldr     r1, [r11, #-12]
   0x0001071c <+44>:    bl      0x1055c
   0x00010720 <+48>:    mov     r3, r0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    ldr     r1, [pc, #48]   ; 0x10760 <main+112>
   0x0001072c <+60>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010730 <+64>:    ldr     r0, [pc, #36]   ; 0x1075c <main+108>
   0x00010734 <+68>:    ldr     r1, [r11, #-8]
   0x00010738 <+72>:    bl      0x1055c
   0x0001073c <+76>:    mov     r3, r0
   0x00010740 <+80>:    mov     r0, r3
   0x00010744 <+84>:    ldr     r1, [pc, #20]   ; 0x10760 <main+112>
   0x00010748 <+88>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x0001074c <+92>:    mov     r3, #0
   0x00010750 <+96>:    mov     r0, r3
   0x00010754 <+100>:   sub     sp, r11, #4
   0x00010758 <+104>:   pop     {r11, pc}
   0x0001075c <+108>:   andeq   r0, r2, r8, asr #19
   0x00010760 <+112>:                   ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

It is clear that the value for the post-decrement operator gets loaded into **r1** at **main+68** so let's break at **main+72**.

```
(gdb) b *main+72
```

We can clearly see that **r1** does in fact hold the value of **15** to which was decremented from our original value.

```
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    sub     r2, r3, #1
   0x0001070c <+28>:    str     r2, [r11, #-8]
   0x00010710 <+32>:    str     r3, [r11, #-12]
   0x00010714 <+36>:    ldr     r0, [pc, #64]   ; 0x1075c <main+108>
   0x00010718 <+40>:    ldr     r1, [r11, #-12]
   0x0001071c <+44>:    bl      0x1055c
   0x00010720 <+48>:    mov     r3, r0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    ldr     r1, [pc, #48]   ; 0x10760 <main+112>
   0x0001072c <+60>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010730 <+64>:    ldr     r0, [pc, #36]   ; 0x1075c <main+108>
   0x00010734 <+68>:    ldr     r1, [r11, #-8]
=> 0x00010738 <+72>:    bl      0x1055c
   0x0001073c <+76>:    mov     r3, r0
   0x00010740 <+80>:    mov     r0, r3
   0x00010744 <+84>:    ldr     r1, [pc, #20]   ; 0x10760 <main+112>
   0x00010748 <+88>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x0001074c <+92>:    mov     r3, #0
   0x00010750 <+96>:    mov     r0, r3
   0x00010754 <+100>:   sub     sp, r11, #4
   0x00010758 <+104>:   pop     {r11, pc}
   0x0001075c <+108>:   andeq   r0, r2, r8, asr #19
   0x00010760 <+112>:                   ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print $r1
$1 = 15
```

Next week we will dive into Hacking Post-Decrement Operator.

Part 49 – Hacking Post-Decrement Operator

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's once again review our code.

```
#include <iostream>

int main(void) {
    int myNumber = 16;
    int myNewNumber = myNumber--;

    std::cout << myNewNumber << std::endl;
    std::cout << myNumber << std::endl;

    return 0;
}
```

Let's review last week's debug.

```
(gdb) disas
Dump of assembler code for function main:
   0x000106f0 <+0>:     push    {r11, lr}
   0x000106f4 <+4>:     add     r11, sp, #4
   0x000106f8 <+8>:     sub     sp, sp, #8
   0x000106fc <+12>:    mov     r3, #16
   0x00010700 <+16>:    str     r3, [r11, #-8]
   0x00010704 <+20>:    ldr     r3, [r11, #-8]
   0x00010708 <+24>:    sub     r2, r3, #1
   0x0001070c <+28>:    str     r2, [r11, #-8]
   0x00010710 <+32>:    str     r3, [r11, #-12]
   0x00010714 <+36>:    ldr     r0, [pc, #64]   ; 0x1075c <main+108>
   0x00010718 <+40>:    ldr     r1, [r11, #-12]
   0x0001071c <+44>:    bl      0x1055c
   0x00010720 <+48>:    mov     r3, r0
   0x00010724 <+52>:    mov     r0, r3
   0x00010728 <+56>:    ldr     r1, [pc, #48]   ; 0x10760 <main+112>
   0x0001072c <+60>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x00010730 <+64>:    ldr     r0, [pc, #36]   ; 0x1075c <main+108>
   0x00010734 <+68>:    ldr     r1, [r11, #-8]
=> 0x00010738 <+72>:    bl      0x1055c
   0x0001073c <+76>:    mov     r3, r0
   0x00010740 <+80>:    mov     r0, r3
   0x00010744 <+84>:    ldr     r1, [pc, #20]   ; 0x10760 <main+112>
   0x00010748 <+88>:    bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
   0x0001074c <+92>:    mov     r3, #0
   0x00010750 <+96>:    mov     r0, r3
   0x00010754 <+100>:   sub     sp, r11, #4
   0x00010758 <+104>:   pop     {r11, pc}
   0x0001075c <+108>:   andeq   r0, r2, r8, asr #19
   0x00010760 <+112>:                   ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print $r1
$1 = 15
```

As we can see here the value in **r1** at **main+68** is **15**. Let's hack!

```
(gdb) set $r1 = 666
(gdb) c
Continuing.
666
[Inferior 1 (process 1017) exited normally]
```

Once again we have manipulated and changed program execution to our own bidding. With each of these bite-size lessons you continue to get a better grasp on the processor and how it interfaces with the binary.

I hope this series gives you a solid framework for understanding the ARM processor. This concludes the series. Thank you all for coming along on the journey!

The x64 Architecture

Let's dive in rightaway!

Part 1 – The Cyber Revolution

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

I often wonder when I see all the latest hacks on a variety of networks, computers and IoT devices how many people really have even the most basic understanding of what goes on down to the microprocessor level.

For years I have published x86 and ARM Assembly and Reverse Engineering tutorials with the intent of opening up the eyes of the public to better understand what Assembly Language is in addition to the notion that there is actually more than just the decimal number system.

Today we have drones, AI, IoT and smart devices that the public rarely understands what the true impact is on their privacy or security.

Everything is Cyber. No matter what you do or where you go or where you live or where you work you will be forced to engage "The Cyber Revolution".

This tutorial series is your opportunity to learn FREE OF CHARGE the very basics of x64 Assembly. Naturally you might ask what is x64 Assembly and why would I possibly want to understand the basics of it let alone Reverse Engineering?

Just about every computer and server today including the cloud runs on an x64 based chipset. Just about every phone, IoT and tablet device runs on an ARM chip (with a number of exceptions). Our last tutorial series dove deep into the ARM chip so if you would like to dive in please review the archives here on my LinkedIn profile.

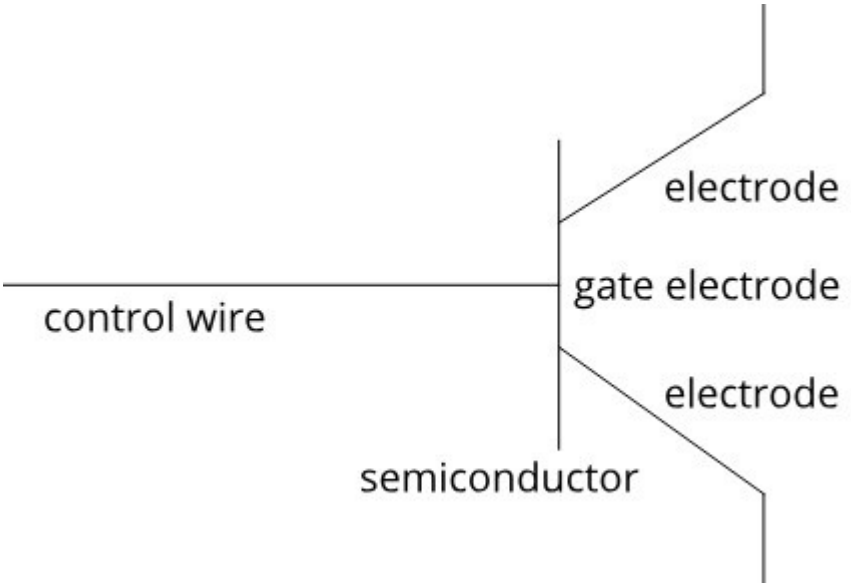
Understanding x64 will give you a better idea of the very infrastructure that supports just about everything we do. You do not have to have any computer science skills to take this FREE course. Simply a few minutes of your time once a week will do.

Let's dive in!

Part 2 - Transistors

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

To understand modern computing we have to go down to the most basic level. Our journey starts with the transistor.



A transistor is nothing more than a complex relay as it is a switch that can be open or closed by applying an electrical charge. This charge is made possible by the use of a control wire. The control wire is attached to a material that can conduct or resist electricity to which on the other end there are two electrodes attached to such a material. This is the concept of a semiconductor. The control wire attaches to a gate electrode where if you change the electrical charge of the gate the conductivity of the semiconductor material can be manipulated. Think of a simple kitchen faucet to which you can turn water on or off. The concept is quite similar.

Quite simply the flow of electricity represents a 1 and the lack of such an electricity flow represents a 0. This is a boolean on or off architecture to which we need to take a deeper dive into the binary number system at a later time.

I deliberately try to keep these lessons short so that it draws the largest audience to take just a few minutes each week to properly grasp some complicated architectures.

Next week we will touch on logic gates and discuss how the combination of such gates make up the core of how the processor works. We will only discuss them on a high level as it would be an entire additional course in electrical engineering to really get into how the processor is made to which we will stick to the basics and spend more of our time on how to program the chip.

After some basics about the processor and an introduction to the binary and hexadecimal number systems we will build our very own bootable operating system.

Part 3 - Logic Gates

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In our last tutorial we spoke briefly about binary to which we represent as either true or false. In binary, true is equal to 1 and false is equal to 0. Computers are ultimately built on this very simple concept to which at the core we have four possible logic gates which can be combined in an infinite amount of sequences.

Let's start with the **AND Gate** below.

output	input 1	input 2
1	1	1
0	1	0
0	0	1
0	0	0

In an AND Gate there are two binary values to which outputs 1 only if both binary values are 1.

The **NOT Gate** is represented below.

output	input
0	1
1	0

In a NOT Gate it simply takes a single binary value and negates it.

The **OR Gate** is represented below.

output	input 1	input 2
1	1	1
1	1	0
1	0	1
0	0	0

In an OR Gate only one of the inputs has to be 1 in order to output a 1.

The **XOR Gate** is represented below.

output	input 1	input 2
0	1	1
1	1	0
1	0	1
0	0	0

In an XOR Gate if both inputs are either 0 or 1 the output is 0.

"The Why..." Ok so why am I going over this? What does this have to do with understanding Assembly or Reverse Engineering? Well... At the very CORE of all processors are these simple logic gates that when combined together form complex instructions. I could spend literally years showing you this in practice

however I will leave that for another to pick up the charge. What is important is that you get a basic understanding of what is going on here when we ultimately see instructions such as AND, OR, XOR, etc when we code in Assembly and more importantly when we Reverse Engineer.

Stay tuned! We will be building our own very SIMPLE Operating System shortly!

Part 4 - Number Systems

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

It really all breaks down to 1 and 0. No matter how sophisticated the future frameworks evolve they all including interpreted languages ultimately use a JVM or the like and go down to Assembly then Machine Code then binary.

Why would we need to even talk about number systems? Why is it relevant to our series here? The answer is simple. In addition to everything going down to 1 and 0, the instructions and memory in addition to the processor registers all utilize another number system called hexadecimal.

Let's discuss binary! At the core of the microprocessor are a series of binary numbers which are either +5V (on or 1) or 0V (off or 0). Each 0 or 1 represents a bit of information within the microprocessor. A combination of 8 bits results in a single byte.

Before we dive into binary, let's examine the familiar decimal. If we take the number 2017, we would understand this to be two thousand and seventeen.

Value	1000s	100s	10s	1s
Representation	10^3	10^2	10^1	10^0
Digit	2	0	1	7

Let's take a look at the binary system and the basics of how it operates.

Bit Number	b7	b6	b5	b4	b3	b2	b1	b0
Representation	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Weight	128	64	32	16	8	4	2	1

If we were to convert a binary number into decimal, we would very simply do the following. Let's take a binary number of 0101 1101 and as you can see it is 93 decimal.

Bit	Weight	Value
0	128	0
1	64	64
0	32	0
1	16	16
1	8	8
1	4	4
0	2	0
1	1	1

Adding the values in the value column gives us 0 + 64 + 0 + 16 + 8 + 4 + 0 + 1 = 93 decimal.

If we were to convert a decimal number into binary, we would check to see if a subtraction is possible relative to the highest order bit and if so, a 1 would be placed into the binary column to which the remainder would be carried into the next row. Let's consider the example of the decimal value of 120 which is 0111 1000 binary.

```
128    64    32    16    8    4    2    1
0      1     1     1     1    0    0    0
```

1)Can 128 fit inside of 120: No, therefore 0.

2)Can 64 fit inside of 120: Yes, therefore 1, then 120 – 64 = 56.

3)Can 32 fit inside of 56: Yes, therefore 1, then 56 – 32 = 24.

4)Can 16 fit inside of 24: Yes, therefore 1, then 24 – 16 = 8.

5)Can 8 fit inside of 8: Yes, therefore 1, then 8 – 8 = 0.

6)Can 4 fit inside of 0: No, therefore 0.

7)Can 2 fit inside of 0: No, therefore 0.

8)Can 1 fit inside of 0: No, therefore 0.

When we want to convert binary to hex we simply work with the following table.

```
Decimal    Hex    Binary
0          0      0000
1          1      0001
2          2      0010
3          3      0011
4          4      0100
5          5      0101
6          6      0110
7          7      0111
8          8      1000
9          9      1001
10         A      1010
11         B      1011
12         C      1100
13         D      1101
14         E      1110
15         F      1111
```

Let's convert a binary number such as 0101 1111 to hex. To do this we very simply look at the table and compare each nibble which is a combination of 4 bits. Keep in mind, 8 bits is equal to a byte and 2 nibbles are equal to a byte.

```
0101 = 5
1111 = F
```

Therefore 0101 1111 binary = 0x5f hex. The 0x notation denotes hex.

To go from hex to binary it's very simple as you have to simply do the opposite such as:

```
0x3a = 0011 1010

3 = 0011
A = 1010
```

It is important to understand that each hex digit is a nibble in length therefore two hex digits are a byte in length.

To convert from hex to decimal we do the following:

```
0x5f = 95

5 = 5 x 16^1 = 5 x 16 = 80
F = 15 x 16^0 = 15 x 1 = 15
```

Therefore we can see that 80 + 15 = 95 which is 0x5f hex.

Finally to convert from decimal to hex. Let's take the number 850 decimal which is 352 hex.

```
Division Result(No Remainder) Remainder Remainder Multiplication
850 / 16 53                   0.125     0.125 x 16 = 2
53 / 16  3                    0.3125    0.3125 x 16 = 5
3 / 16   0                    0.1875    0.1875 x 16 = 3
```

"Why the hell would I waste my time learning all this crap when the computer does all this for me!"

As I mentioned above, it is vital you have a good understanding of these two additional number systems if you are truly to grasp and master reverse engineering at its core. There are some amazing tools that help the RE process however the better understanding that you have of these will help you as you grow.

I am not suggesting you memorize the above, nor am I suggesting that you do a thousand examples of each. All I ask is that you take the time to really understand that literally everything and I mean everything goes down to binary bits in the processor.

Whether you are creating, debugging or hacking an Assembly, Python, Java, C, C++, R, JavaScript, or any other new language application that hits the street, ultimately everything MUST go down to binary 0 and 1 to which represent a +5V or 0V.

We as humans operate on the base 10 decimal system. Let's expand our mind to base 2 binary and base 16 hexadecimal!

Next week we will dive into binary addition! Stay tuned!

Part 5 - Binary Addition

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Binary addition can occur in one of four different fashions:

```
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0 (1) [One Plus One Equals Zero, Carry One]
```

Keep in mind the (1) means a carry bit. It very simply means an overflow.

Lets take the following 4-bit nibble example:

```
  0111
+ 0100
= 1011
```

We see an obvious carry in the 3rd bit. If the 8th bit had a carry then this would generate a carry flag within the CPU.

Let's examine an 8-bit number:

```
  01110000
+ 01010101
= 11000101
```

If we had:

```
    11110000
+   11010101
= (1)11000101
```

Here we see a carry bit which would trigger the carry flag within the CPU to be 1 or true. We will discuss the carry flag in later tutorials. Please just keep in mind this example to reference as it is very important to understand.

Next week we will dive into binary subtraction! Stay tuned!

Part 6 - Binary Subtraction

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Binary subtraction is nothing more than adding the negative value of the number to be subtracted. For example 8 + - 4, the starting point would be zero to which we move 8 points in the positive direction and then four points in the negative direction yielding a value of 4.

We represent a sign bit in binary to which bit 7 indicates the sign of number where 0 is positive and 1 is negative.

```
Sign Bit 7     Bits 0 — 6
1              0000011
```

The above would represent -2.

We utilize the concept of twos compliment which inverts each bit and then finally adding 1.

Lets example binary 2.

```
00000010
```

Invert the bits.

```
11111101
```

Add 1.

```
  11111101
+ 00000001
  11111110
```

Let's examine a subtraction operation:

```
   00000100  4 decimal
 + 11111110 -2 decimal
(1)00000010  2 decimal
```

So what is the (1) you may ask, that is the overflow bit. In future tutorials we will examine what we refer to as the overflow flag and carry flag.

Next week we will dive into word lengths! Stay tuned!

Part 7 - Word Lengths

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Before we dive into the architecture lets talk about how we define various bits and how they are structured within the processor.

In both x64 and x86, we define a byte as 8 bits. We define a word as 16 bits. We define a double word as 32 bits and a quadword as 64 bits. Finally we define a double quadword as 128 bits.

Intel processors store bytes as what we refer to as "little endian," meaning lower significant bytes are stored in lower memory addresses. Lets give an example of a simple 16-bit or 2 byte value. On disk - 0xAABB. When it goes into memory it is stored as 0xBBAA as I hope this provides a good visual as this concept can be quite confusing.

Keep in mind, 8 bits make up a byte. 4 bits are also called a nibble which are equivalent to one hex digit.

Next week we will dive into general architecture! Stay tuned!

Part 8 - General Architecture

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The x64 architecture is a backwards-compatible extension of the x86 platform. It provides a legacy 32-bit mode, which is identical to x86, and a new 64-bit mode. You can review my legacy x86 tutorial if you would like to get more information right here on LinkedIn.

The term "x64" includes both AMD 64 and Intel64. The instruction sets are similar.

x64 extends x86's 8 general-purpose registers to be 64-bit, and adds 8 new 64-bit registers. The 64-bit registers have names beginning with "**r**", so for example the 64-bit extension of **eax** is called **rax**. The new registers are named **r8** through **r15**.

The lower 32 bits, 16 bits, and 8 bits of each register are directly addressable in operands. This includes registers, like **esi**, whose lower 8 bits were not previously addressable. The following table specifies the assembly-language names for the lower portions of 64-bit registers.

The table below breaks out each bytes distinction.

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Operations that output to a 32-bit subregister are automatically zero-extended to the entire 64-bit register. Operations that output to 8-bit or 16-bit subregisters are *not* zero-extended (this is compatible x86 behavior).

The high 8 bits of **ax**, **bx**, **cx**, and **dx** are still addressable as **ah**, **bh**, **ch**, **dh**, but cannot be used with all types of operands.

The instruction pointer, **eip**, and **flags** register have been extended to 64 bits (**rip** and **rflags**, respectively) as well.

The x64 processor also provides several sets of floating-point registers:

- Eight 80-bit x87 registers.
- Eight 64-bit MMX registers. (These overlap with the x87 registers.)
- The original set of eight 128-bit SSE registers is increased to sixteen.

Next week we will dive into calling conventions! Stay tuned!

Part 9 - Calling Conventions

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The x64 processor uses what we refer to as __fastcall.

The __fastcall calling convention specifies that arguments to functions are to be passed in registers, when possible. This calling convention only applies to the x86 architecture.

The first two DWORD or smaller arguments that are found in the argument list from left to right are passed in **ecx** and **edx** registers; all other arguments are passed on the stack from right to left.

Called function pops the arguments from the stack.

At sign (@) is prefixed to names; an at sign followed by the number of bytes (in decimal) in the parameter list is suffixed to names.

No case translation performed.

Here is a simple breakdown to illustrate:

Parameter	QWORD	DWORD	WORD	BYTE
1	rcx	ecx	cx	cl
2	rdx	edx	dx	dl
3	r8	r8d	r8w	r8b
4	r9	r9d	r9w	r9b
4+	stack	stack	stack	stack

If you have two parameters you are passing from a function, for example int **x** and int **y** and it is a QWORD, **x** will go into **rcx** and **y** will go into **rdx**.

If you have five parameters you are passing for example int **a**, int **b**, int **c**, int **d**, int **e** and it is a WORD in length, **a** will go into **cx**, **b** into **dx**, **c** into **r8w**, **d** into r9w and e into the stack.

Next week we will dive into boolean instructions! Stay tuned!

Part 10 - Boolean Instructions

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

There are four boolean instructions to which exist are AND, OR, XOR and NOT. Earlier in this tutorial we briefly discussed gates which took advantage of the same logic down to the metal. We will see this logic throughout our reversing so it is important to understand what it does down at the individual bit level.

AND = If the first number has a 0 and the second number has a 0, the result is 0.

AND = If the first number has a 0 and the second number has a 1, the result is 0.

AND = If the first number has a 1 and the second number has a 0, the result is 0.

AND = If the first number has a 1 and the second number has a 1, the result is 1.

ex: 0 0 1 0 0 0 1 0

ex: 0 1 1 0 1 1 1 0

ex:————————

ex: 0 0 1 0 0 0 1 0

OR = If the first number has a 0 and the second number has a 0, the result is 0.

OR = If the first number has a 0 and the second number has a 1, the result is 1.

OR = If the first number has a 1 and the second number has a 0, the result is 1.

OR = If the first number has a 1 and the second number has a 1, the result is 1.

ex: 0 0 1 0 0 0 1 0

ex: 0 1 1 0 1 1 1 0

ex:————————

ex: 0 1 1 0 1 1 1 0

XOR = If the first number has a 0 and the second number has a 0, the result is 0.

XOR = If the first number has a 0 and the second number has a 1, the result is 1.

XOR = If the first number has a 1 and the second number has a 0, the result is 1.

XOR = If the first number has a 1 and the second number has a 1, the result is 0.

ex: 0 0 1 0 0 0 1 0

ex: 0 1 1 0 1 1 1 0

ex:————————

ex: 0 1 0 0 1 1 0 0

NOT = If the first number has a 0 the second number becomes 1.

NOT = If the first number has a 1 the second number becomes 0.

ex: 0 0 1 0 0 0 1 0

ex:——————

ex: 1 1 0 1 1 1 0 1

Next week we will dive into pointers! Stay tuned!

Part 11 - Pointers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

x64 utilizes the flat memory model to which we have one large array of addresses that exist within the processor.

A pointer is nothing more than the address of a specific value in memory. Let's take an example:

mov rax, 0x10

In this example we are moving **10** hex into the **rax** register.

To get the value inside **rax** at **0x10,** you would use the following syntax:

mov rbx, word ptr [rax]

Let's assume the value inside memory **0x10** was **0x20** therefore **rax** points to the value inside **0x10** which when you dereference by **[rax]** contains **0x20**. **0x20** is the value inside of the register **rax**.

We are moving a word value pointed inside of **rax** into **rbx**.

If we do:

mov word ptr [rax], 0x66

This will put the value of **0x66** into the memory location at **0x10**. We know that the value inside **0x10** memory location was **0x20** so therefore the new value inside the memory at **0x10** will be **0x66.**

This can get confusing however when we get into code over the coming months this will become more apparent.

Next week we will dive into load effective address! Stay tuned!

Part 12 - Load Effective Address

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

When a binary executes in RAM the OS will unmap the code into a data segment where it finds free space in memory.

Load Effective Address loads a given memory address as a pointer to any given variable. For example:

lea rbx, my_var

This will load the address of **my_var** into **rbx**.

In C++, a pointer actually adds what the user would see as one if something was incremented however it is actually moving it 2 bytes forward under the hood assuming it is a word in length or 16 bits or 2 bytes. Same thing.

In Assembly every single byte is addressable. For example:

lea rax, my_var

inc rax

mov word ptr [rax], rbx

Let's say the value of **0x20** is in **rbx**. This above instruction will place the value of **0x20** into a non-word boundary which will result in an error. You would have to increment **rax** by 2 to ensure that does not happen.

Next week we will dive into the data segment! Stay tuned!

Part 13 - The Data Segment

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The data segment allocates memory on the heap in memory rather than the stack as they are not local variables they are known throughout the entire binary.

The sizes of data are as follows:

1)byte - We use the db notation which is obviously 1 byte or 8 bits.

2)word - We use dw and it is 2 bytes in length.

3)double word - We use dd to assign and they are 4 bytes long.

4)quad word - We use dq which is 8 bytes long.

5)xmm word - We use xmmword which is 16 bytes long.

6)ymm word - We use ymmword which is 32 bytes long.

There are SSE math registers which are separate from the CPU which hold the following:

1)real4 - This is a single or what you would think of as a floating point numbers as this is 4 bytes long.

2)real8 - This is a double floating point as this is 8 bytes long.

Finally there are arrays which can be single or multidimensional arrays where you can allocate against a db, dw, dd, dq, xmmword or ymmword.

We will see this in code when we get more advanced into the series however its critical that you understand the variables within a function are local and go to the stack as they do not last throughout the program. These variables which are part of the data segment are not local they are global and go to the heap.

The stack - local vars - grows down in memory so they start at a high memory address and grow down. The heap - global vars - grows from a lower memory address and grows up.

If you have questions please ask them in the comments as it is critical you get this concept down when we start to build our very basic operating system.

Next week we will dive into SHL! Stay tuned!

Part 14 - SHL Instruction

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The SHL command stands for shift left.

Let's assume the register **al** holds **01010101b** which is an 8-bit binary value. Let's assume the instruction is **shl al, 2**. Below is what transpires as we see the values move two bits to the left.

00010101

00010101

Therefore the new value will be:

10100000

Next week we will dive into SHR! Stay tuned!

Part 15 - SHR Instruction

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The SHR command stands for shift right.

Let's assume the register **al** holds 00010100b which is an 8-bit binary value. Let's assume the instruction is **shr al, 2**. Below is what transpires as we see the values move two bits to the left.

00010100

00010100

```
1 section .data
2
3 section .text
4   global _start
5
6 _start:
7   mov al, 00010100b
8   shr al, 2
9   nop
pc@pc-mytechnotalent:~/Desktop$ nasm -f elf64 -o test.o test.asm
pc@pc-mytechnotalent:~/Desktop$ ld -o test test.o
pc@pc-mytechnotalent:~/Desktop$ gdb -q test
Reading symbols from test...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x400080
(gdb) set disassembly-flavor intel
(gdb) r
Starting program: /home/pc/Desktop/test

Breakpoint 1, 0x0000000000400080 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000400080 <+0>:     mov    al,0x14
   0x0000000000400082 <+2>:     shr    al,0x2
   0x0000000000400085 <+5>:     nop
End of assembler dump.
(gdb) si
0x0000000000400082 in _start ()
(gdb) si
0x0000000000400085 in _start ()
(gdb) p /t $al
$1 = 101
```

00000101

Next week we will dive into ROL! Stay tuned!

Part 16 - ROL Instruction

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The ROL command stands for rotate left.

```
1 section .text
2   global _start
3
4 _start:
5   mov al, 00000001b
6   rol al, 0x01
7   nop
```

In our simple x64 example on an Ubuntu Linux machine above we see we **mov 1** into **al** and rotate left by 1 bit.

The binary representation is **00000001b**. If we **ROL** 1 bit the value simply becomes **00000010b** as demonstrated below.

We first compile and link by:

nasm -f elf64 -o test.o test.asm

ld -o test test.o

```
pc@pc-mytechnotalent:~/Downloads$ gdb -q test
Reading symbols from test...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x400080
(gdb) r
Starting program: /home/pc/Downloads/test

Breakpoint 1, 0x0000000000400080 in _start ()
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000400080 <+0>:     mov    al,0x1
   0x0000000000400082 <+2>:     rol    al,1
   0x0000000000400084 <+4>:     nop
End of assembler dump.
(gdb) si
0x0000000000400082 in _start ()
(gdb) p /t $al
$1 = 1
(gdb) si
0x0000000000400084 in _start ()
(gdb) p /t $al
$2 = 10
```

We can see here in the debugger that **al** starts with **1** and when we rotate left it goes to **10b**.

You can ROL with additional bits as well. The logic would remain the same as the bits will rotate left just as we demonstrated above.

Next week we will dive into ROR! Stay tuned!

Part 17 - ROR Instruction

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

The ROR command stands for rotate right.

```
1 section .text
2   global _start
3
4 _start:
5   mov al, 00000001b
6   ror al, 0x01
7   nop
```

In our simple x64 example on an Ubuntu Linux machine above we see we mov 1 into al and rotate right by 1 bit.

The binary representation is **00000001b**. If we **ROR** 1 bit the value simply becomes **10000000b** as demonstrated below.

We first compile and link by:

nasm -f elf64 -o test.o test.asm

ld -o test test.o

```
pc@pc-mytechnotalent:~/Documents$ gdb -q test
Reading symbols from test...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x400080
(gdb) set disassembly-flavor intel
(gdb) r
Starting program: /home/pc/Documents/test

Breakpoint 1, 0x0000000000400080 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000400080 <+0>:     mov    al,0x1
   0x0000000000400082 <+2>:     ror    al,1
   0x0000000000400084 <+4>:     nop
End of assembler dump.
(gdb) si
0x0000000000400082 in _start ()
(gdb) si
0x0000000000400084 in _start ()
(gdb) p /t $al
$1 = 10000000
```

We can see here in the debugger that al starts with 1 and when we rotate right it goes to **10000000b**.

Next week we will dive into Boot Sector Basics! Stay tuned!

Part 18 - Boot Sector Basics [Part 1]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Over the next few tutorials we are going to write a very basic x86 Operating System to which we will use QEMU which is a full system emulator or OS emulator. You could also install VirtualBox and ultimately convert our boot loader to an ISO if you so choose.

At the very core of a computer booting is what we refer to as the boot loader. The boot loader physically reads the first sector or sector 0 from your HD or other media to ultimately bootstrap an OS.

When the computer boots it reads the first sector which is exactly 0x200 bytes (hex) or 512 bytes in decimal.

The system that is reading this boot loader is what is referred to as BIOS which is a basic input output system and it loads in 16-bit mode. It does this to be compatible with older processors. Modern processors immediately switch to what we refer to as UEFI which is a more sophisticated IO system however we will focus on the very basics here with BIOS.

Next week we will discuss what exactly goes on when BIOS reads the boot sector.

Part 19 - Boot Sector Basics [Part 2]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

We are at the stage where we are going to start integrating real-world code. If you do not have an active linux desktop I would suggest you get Virtualbox and Ubuntu on either your Windows or Mac. I have a prior tutorial that will walk you through this process below. For some reason I am not able to embed the link so please just copy and paste it into your browser.

https://www.linkedin.com/pulse/assembly-language-basic-malware-reverse-engineering-kevin-m-thomas-16/

You will additionally need a text editor for the terminal. I use VIM. You will find a link to set that up as well below.

https://www.linkedin.com/pulse/assembly-language-basic-malware-reverse-engineering-kevin-m-thomas-17/

In addition you will have to install nasm so you may simply type:

sudo apt-get install nasm

NASM is the assembler we will use and we will focus on the intel syntax. First go into the terminal and fire up Vim and type the following:

```
1 loop:
2   jmp loop
```

Remember to type 'i' to insert and then 'esc' and 'wq' to go into command mode and save your file.

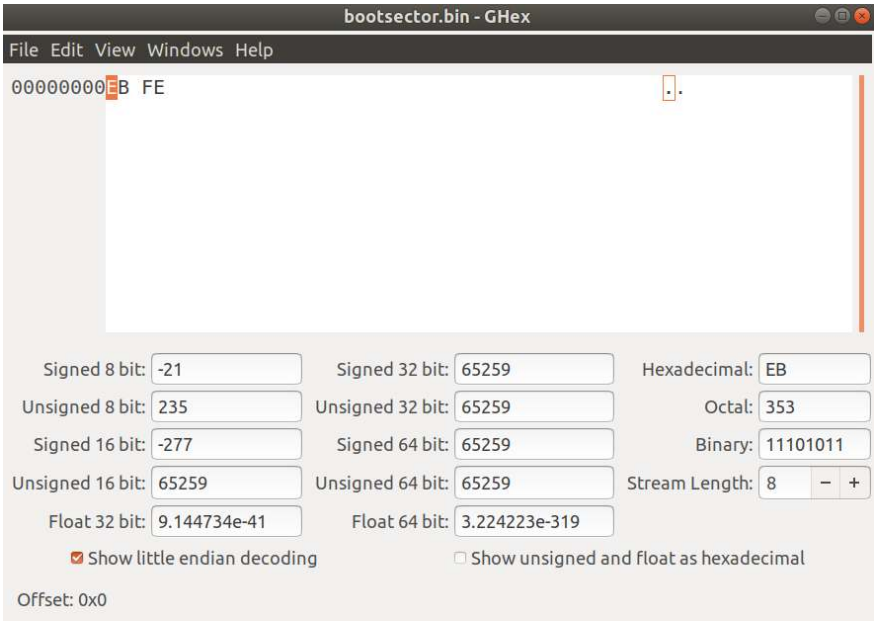
The above line simply sets an infinite loop and does nothing more. The **loop** label is created to which we simply **jmp** back to itself. This code in itself will compile however it will not run in an OS as it does not trigger what we refer to as the magic number to which BIOS looks to understand this is the end of your boot sector. We will cover more on that in future lectures.

```
nasm bootsector.asm -f bin -o bootsector.bin
```

We type the above command assuming you saved your file in vim as **bootsector.asm**. This will create a binary file to which we will examine the contents within a hex editor. A hex editor is an application that examines each byte of data that is compiled into a file. We will see that our assembly instructions above will ultimately get translated down to their raw opcode values. The processor only understands raw opcodes which are simply operation codes. Below is a link to a table identifying the opcodes. I saved you the effort of referencing the intel dataset as it is literally thousands of pages and several volumes:

http://ref.x86asm.net/coder64.html

Let's use a hex editor like ghex and open up our bin file.



We see **EB FE** which are hex bytes and each letter is a nibble (a nibble is 4 bits or half a byte). Both **EB FE** make up two full bytes. Keep in mind the processor reads from disk in reverse byte order such that **FE** gets read first and then **EB**. This process is called little endian and is how the x64 processor works.

If you review the table to which I provided the link you will see that **FE** represents an **INC** or increment by one. This is our loop value.

Next you will find that **EB** stands for **JMP** which is our jump instruction above.

This is alot of information if you are new to assembly. Take it step-by-step and follow along with me in a real linux OS and with each lesson you will get a better understanding of the basics.

Next week we will build upon this lesson by adding some simple data to our binary.

Part 20 - Boot Sector Basics [Part 3]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

For those of you that are familiar with assembly these next several weeks/months might seem like we are progressing very slowly however the aim is to help those with little understanding of hardware to get a better understanding of the very systems that power the cloud.

The vast majority of AWS and Azure as well as many other cloud services utilize x64 based operating systems. Understanding what happens when these systems boot is of significant value and that is why we are going to go thorough a very slow process looking at each piece of a boot sector when a machine loads.

Let's examine our source code. Follow along in Vim or Nano.

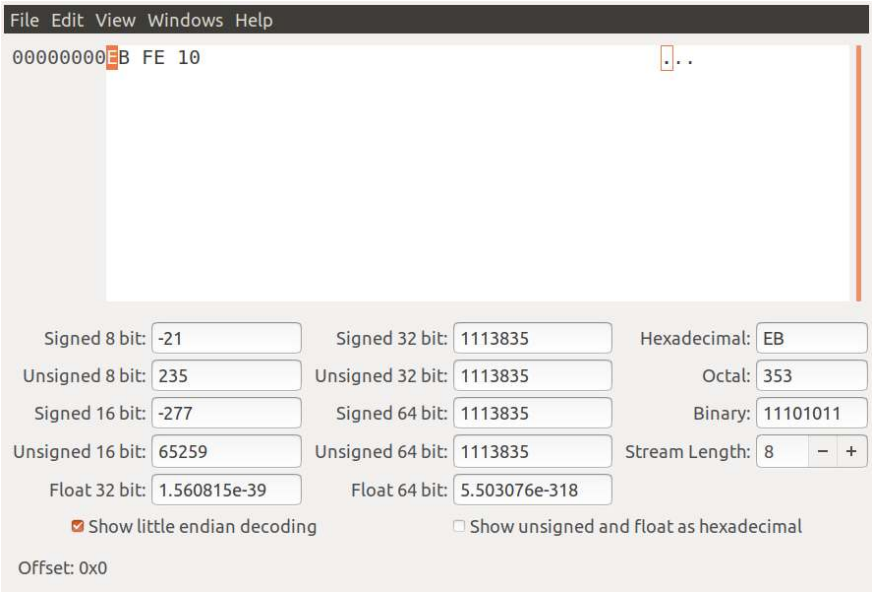
```
1 loop:
2   jmp loop
3
4 db 0x10
```

Last week we learned the opcodes for line 1 and 2 to which we do not have to review. Today we add a byte of data into our code. Notice this is a hexadecimal number and will match our binary upon inspection. In future lessons we will see how it looks when we do decimal and other systems.

Let's compile. If you do not have NASM installed please ensure you type **sudo apt-get install nasm**.

```
nasm bootsector.asm -f bin -o bootsector.bin
```

Let's look at our binary in a hex editor. I use GHex as I keep to the GNU tradition as we will in future lessons use the GNU debugger called GDB. These are all on your Linux systems as I am using Ubuntu for these tutorials.



We saw last week that the **EB** and **FE** correspond to our **INC** and **JMP** instructions. If this is unclear please re-read last weeks lecture. We see the 3rd byte as **10**. Remember this is hexadecimal so the value in decimal would be **16**.

Next week we will keep adding to our code and progress in our OS development series.

Part 21 - Boot Sector Basics [Part 4]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Today we continue our Boot Sector Basics. Let's examine the code:

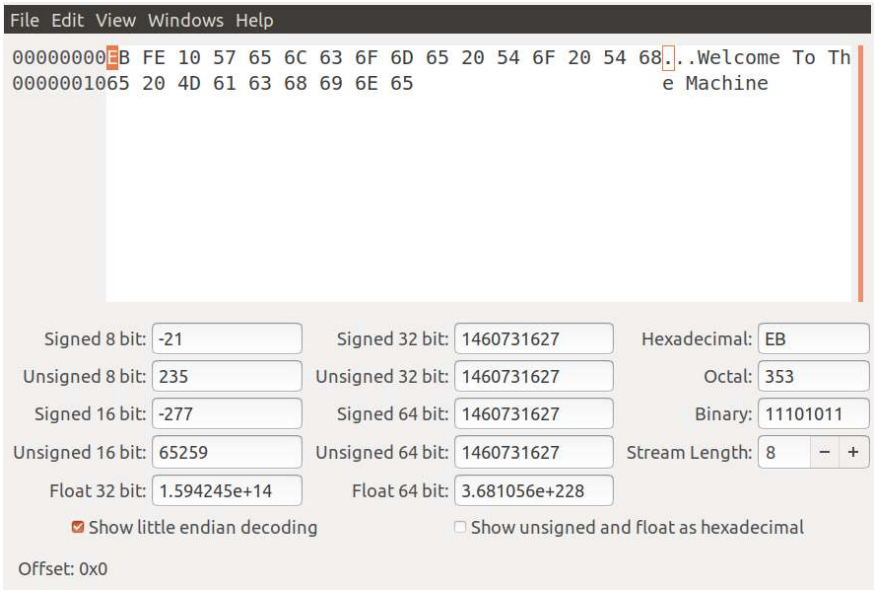
```
loop:
  jmp loop

db 0x10
db 'Welcome To The Machine'
```

We add a string to our code as seen above and compile.

```
nasm bootsector.asm -f bin -o bootsector.bin
```

Let's examine the binary in a hex editor.



Closely examine the above. We see our original code which we do not have to review however now we see a series of numbers, hex numbers that represent ASCII characters. We see that each letter corresponds with a letter. When we say that ultimately everything goes down to 0 and 1 this is a proof of concept. As you can see **EB** is selected above and we can see those hex values ultimately go to **11101011** in binary.

Homework: Google and research the ASCII conversion table and do some research on your own and better understand how hex values represent characters.

Next week we take it to the next level. Stay tuned!

Part 22 - Boot Sector Basics [Part 5]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

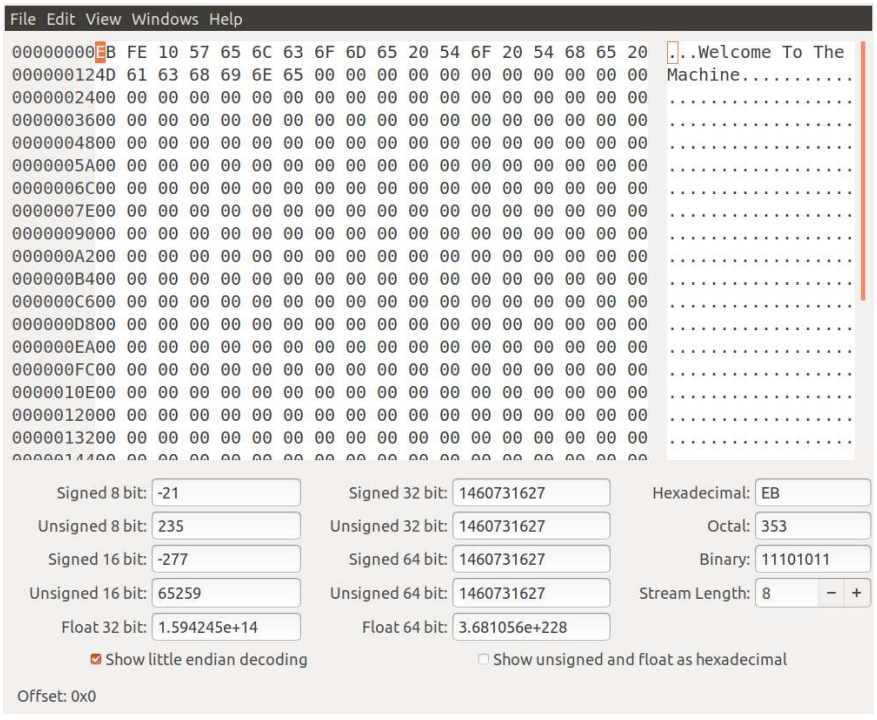
We begin by looking at some simple additions to our code. What we will accomplish today is to create a simple operating system that does literally nothing but boot. We will use QEMU as an emulator as I am too lazy to set up VirtualBox or VMWare however you can easily port the .bin to an .iso if you chose and boot from either.

```
1 loop:
2   jmp loop
3
4 db 0x10
5 db 'Welcome To The Machine'
6
7 times 0x1fe-($-$$) db 0
8 dw 0xaa55
```

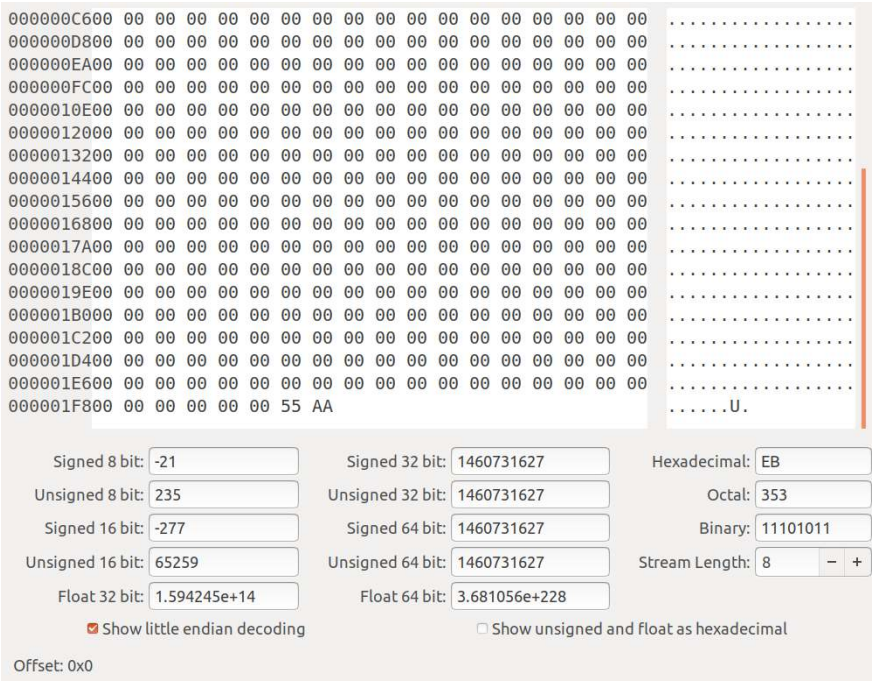
We are simply adding a padding algorithm on line 7 that simply examines how many bytes are left after we subtract 200h or 512 and then it pads the remaining bytes with zeros. At the end you will see what we refer to as the magic number which is **0xaa55** as this is a signature that the cpu is looking for to identify a boot sector. Remember this code is at sector 0 when it boots as there is no file system so if it finds the successful signature it will attempt to boot it.

```
nasm bootsector.asm -f bin -o bootsector.bin
```

We build the binary with the code above. Now let's look at the code in the hex editor.



As you can see it pads out the remaining bytes up to 200h or 512 with 0's as we anticipated. Below is the remainder of the binary.



As you can see at the very end we have **55 AA**. We remember that our processor is little endian so when we code it it was **aa 55** and which is in it's mapped format. When it goes into the cpu it reverses the byte order. This is critical that you understand this.

Next week we will simply do nothing more than launch our new operating system. Stay tuned.

Part 23 - Boot Sector Basics [Part 6]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

This week we will focus on how to use QEMU which is an emulator to boot our simple new OS.

```
sudo apt-get install qemu-system-x86
```

Type the above to obtain qemu specifically for x86 systems.

```
qemu-system-x86_64 bootsector.bin
```

Run the emulator with our binary.

```
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 C980


Booting from Hard Disk...
```

You will see the following. Keep in mind it does nothing but an infinite loop jump which we discussed in detail in previous lessons. This however is the most basic x86 OS one can create.

It simply looks for the signature which we spoke of last week (if this does not make sense please review last weeks lecture) and if it is exactly 200h bytes and it is placed at the first sector of the boot medium the process will be successful.

If you are interested there are different emulators for different architectures.

```
qemu-system-arm     qemu-system-misc    qemu-system-sparc
qemu-system-common  qemu-system-ppc     qemu-system-x86
qemu-system-mips    qemu-system-s390x
```

Next week we will discuss memory addressing so that we can set up a stack within our simple os.

Part 24 - Boot Sector Basics [Part 7]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

We need to discuss memory at this point. Before we can discuss setting up a simple stack in our bootloader we must understand how memory is allocated in the bootsector.

1)**0x0 = Interrupt Vector Table** - This is where our interrupt table exists at the very base of memory. This is where all of our interrupt calls exist.

2)**0x400 = BIOS Data Area** - This stores variables about the state of the bootable device.

3)**0x7c00 = Loaded Boot Sector** - This has our machine code that will be loaded into RAM by the bootloader firmware (note: firmware is simply code that runs before an OS runs like what we are doing).

4)**0x7e00 = Free** - This is your stack area that you can develop in.

5)**0x9fc00 = Extended BIOS Data Area** - Holds data from disk track buffers and other connected devices as remember there is no file system as of yet.

6)**0xa0000 = Video Memory** - BIOS maps your video memory here at boot.

7)**0xc0000 = BIOS** - Where BIOS officially resides.

8)**0x100000 = Free** - Additional space you can develop in.

This is critical that you understand how memory is laid out at boot. In our next lesson we will create a simple stack at **0x7e00**.

Part 25 - Boot Sector Basics [Part 8]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Today we will put all the pieces together. We will create our custom OS that does nothing but boot-up, set a video mode and then only accept numeric digits in the console. This is the final tutorial in this mini-series of Boot Sector Basics.

Let's examine our code:

```
[org 0x7c00]                        ; jmp origin of mapped bootsector code

mov bp, 0xffff                      ; set the stack base
mov sp, bp                          ; set the top of the stack to base

call set_video_mode

call get_char_input

jmp $                               ; iloop so CPU dosen't exec random code

set_video_mode:
    mov al, 0x03                    ; 80x25 8x8 640x200 16 gray
    mov ah, 0x00
    int 0x10
    ret

get_char_input:
    xor ah, ah                      ; wait for a character
    int 0x16

    cmp al, 0x30                    ; compare if numeric 0
    jl get_char_input               ; jmp if less than and restart loop
    cmp al, 0x39                    ; compare if numeric 9
    jg get_char_input               ; jmp if greater than and restart loop

    mov ah, 0x0e
    int 0x10

    jmp get_char_input              ; re-start internal subroutine

times 0x1fe-($-$$) db 0
dw 0xaa55
```

The first thing we do is move to the programable area of the boot sector code at address 0x7c00. We then set the stack base and identify the area for our stack and set the base pointer into the stack pointer.

We then call our video mode function where we set a 640x200 greyscale console. We then call our get character input function that will only allow digits 0 to 9 as you can see 0x30 is the hex ascii value for 0 and 0x39 is the hex ascii value of 9. If the user types anything else in the console literally nothing will enter into the console. This is the absolute control you have in Assembly.

Lets compile and run:

```
nasm bootsector.asm -f bin -o bootsector.bin
qemu-system-x86_64 bootsector.bin
```

We then see the qemu console:



As you can see I am only able to type numeric digits in our OS. Try it for yourself. Write the code and compile and run in the qemu editor. If you do not have qemu installed I show you in detail how to install it in the last two tutorials.

Take the time to really review what I am doing here as it is critical to understand that this is how your computer boots before going into 32 then 64-bit mode.

Next week we will simply discuss the high-level concept of how your computer bridges a 64-bit OS.

Part 26 - Boot Sector Basics [Part 9]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Before we dive into x64 Assembly I want to talk very briefly about what we refer to as long mode.

When the computer boots it needs to enable what we refer to as the A-20 line. In early architectures, processors had 20 address lines which were A-0 to A-19 to which could access 2 to the power of 20 bytes of information. The A-20 line is an external memory reference containing a 16-bit offset address added to a 16-bit segmented number which shifts 4 bits to get the additional access.

This process combined with the Global Descriptor Table allows you to work with your Control Register to to execute a far jump to enter protected mode which is 32-bits.

Long mode which is 64-bit mode which we are all familiar with in our modern architectures extend the address space to access 0xFFFFFFFFFFFFFFFF.

This topic alone can take weeks to explain however I wanted to at a very high level touch base on the fact that the processor needs to bridge to 32-bit mode and then finally to 64-bit through setting the A-20 line, working with the control register and GDT in combination with paging.

I took several months to get to this point so that you have a basic understanding of Assembly as we will start to get into actual 64-bit Assembly in the following tutorials and then our C++ tutorial to which we will reverse engineer each code block into 64-bit Assembly.

Part 27 - x64 Assembly [Part 1]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Today we begin our actual x64 code basics. Over the next few weeks I will create very simple examples so we get a grasp of the x64 architecture. Let's start with a basic code block:

```
section .data

section .text
  global _start

_start:
  mov rax, 0x10

exit:
  mov rax, 60
  mov rdi, 0
  syscall
```

We begin by declaring the **.data** section to which all of our global data is stores. If we had a string or some other form of hard coded data it would go in that block. In our example we will leave it empty.

The **.text** section declares where the entry point of the program will begin in our case we use **_start** or you can use **main**.

We simply move the value of decimal 16 or hex 10 into the 64-bit RAX register. We will see in a moment that the processor will use only the lower EAX when we debug in GDB.

The last piece is just a simple exit routine which we move 60 into RAX and then syscall. It simply returns operation back to the OS.

Let's compile and link:

```
nasm -f elf64 -o 1.o 1.asm
ld 1.o -o 1
```

Let's debug in GDB:

```
gdb -q 1
```

Let's set the debugger for intel syntax and set a break on start:

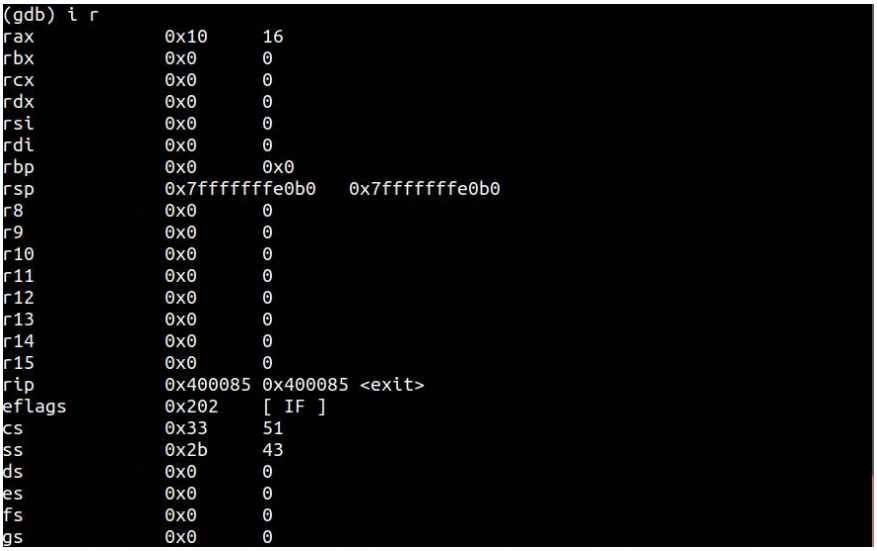
```
Reading symbols from 1...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) b _start
Breakpoint 1 at 0x400080
(gdb) r
Starting program: /home/pc/Desktop/1

Breakpoint 1, 0x0000000000400080 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000400080 <+0>:     mov    eax,0x10
End of assembler dump.
```

As we can see 16 decimal or hex 10 is about to be moved into EAX but as we can see it has not been completed until we step forward.

```
(gdb) si
0x0000000000400085 in exit ()
```

Now we can view our registers.



We can see that RAX holds decimal 16 or hex 10 successfully.

We will spend several weeks on these simple examples so you can get comfortable with how the processor operates and its internal workings.

Part 28 - x64 Assembly [Part 2]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's continue with another example:

```
 1 section .data
 2
 3 section .text
 4   global _start
 5
 6 _start:
 7   mov rax, 0x10
 8   add rax, 0x05
 9
10 exit:
11   mov rax, 60
12   mov rdi, 0
13   syscall
```

As we can see we are moving **0x10** into **RAX** and adding **0x05** into **RAX**.

```
pc@pc-mytechnotalent:~/Desktop$ vim 1.asm
pc@pc-mytechnotalent:~/Desktop$ nasm -f elf64 -o 1.o 1.asm
pc@pc-mytechnotalent:~/Desktop$ ld 1.o -o 1
pc@pc-mytechnotalent:~/Desktop$ gdb -q 1
Reading symbols from 1...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x400080
(gdb) r
Starting program: /home/pc/Desktop/1

Breakpoint 1, 0x0000000000400080 in _start ()
(gdb)
```

We compile and let's disassemble.

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000400080 <+0>:     mov    eax,0x10
   0x0000000000400085 <+5>:     add    rax,0x5
End of assembler dump.
```

As you can see as expected we see our code in debug.

```
(gdb) si
0x0000000000400085 in _start ()
(gdb) si
0x0000000000400089 in exit ()
```

We step twice and then...

```
rax            0x15     21
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x0      0
rbp            0x0      0x0
rsp            0x7fffffffe0b0   0x7fffffffe0b0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x400089 0x400089 <exit>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
```

We see **0x15** or **21** decimal moved into **RAX**. Take the time to carefully try these very simple examples as we go forward.

Part 29 - x64 Assembly [Part 3]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Today we continue our tutorial with a simple subtract example. Let's examine the source code:

```
section .data

section .text
  global _start

_start:
  mov rax, 0x10
  sub rax, 0x05

exit:
  mov rax, 60
  mov rdi, 0
  syscall
```

Let's compile and run the debugger:

```
pc@pc-mytechnotalent:~/Desktop$ nasm -f elf64 -o 1.o 1.asm
pc@pc-mytechnotalent:~/Desktop$ ld 1.o -o 1
pc@pc-mytechnotalent:~/Desktop$ gdb -q 1
Reading symbols from 1...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x400080
(gdb) set disassembly-flavor intel
```

Let's run and disassemble:

```
(gdb) r
Starting program: /home/pc/Desktop/1

Breakpoint 1, 0x0000000000400080 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000400080 <+0>:     mov    eax,0x10
   0x0000000000400085 <+5>:     sub    rax,0x5
End of assembler dump.
```

As we can see very we load **16** or **0x10** hex into **EAX** and then subtract **5** from it in the next instruction.

```
(gdb) si
0x0000000000400085 in _start ()
(gdb) si
0x0000000000400089 in exit ()
```

We step twice and then look at the resulting value in **RAX**.

```
(gdb) i r
rax            0xb      11
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x0      0
rbp            0x0      0x0
rsp            0x7fffffffe0b0   0x7fffffffe0b0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x400089 0x400089 <exit>
eflags         0x212    [ AF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

As we can see the result is 0xb hex or 11 decimal as expected. It is important that you try these simple examples to get a grasp of what happens when we start to debug C++ code in future tutorials.

Part 30 - x64 Assembly [Part 4]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Today we will code our simple, "hello world" program in x64 Assembly.

```
section .data
  text db "Hello World!", 0xA

section .text
  global _start

_start:
  mov rax, 1                    ; write system call
  mov rdi, 1
  mov rsi, text
  mov rdx, 13
  syscall

exit:
  mov rax, 60
  mov rdi, 0
  syscall
```

We simply create a string in the **.data** section and add a return character at the end of the statement. We then perform a simple write call which utilizes the OS's interrupt vector table to spit out our string in the standard output or terminal.

We will compile and run below:

```
pc@pc-mytechnotalent:~/Desktop$ nasm -f elf64 -o 1.o 1.asm
pc@pc-mytechnotalent:~/Desktop$ ld 1.o -o 1
pc@pc-mytechnotalent:~/Desktop$ ./1
Hello World!
```

As we can see "**Hello World**!" has been echoed to the terminal. Next week we will debug this simple program in GDB.

Part 31 - x64 Assembly [Part 5]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

This lecture will be a bit longer than most however it is important that you all take the time to really code and practice the topics discussed below. Let's review our code:

```
section .data
  text db "Hello World!", 0xA

section .text
  global _start

_start:
  mov rax, 1                    ; write system call
  mov rdi, 1
  mov rsi, text
  mov rdx, 13
  syscall

exit:
  mov rax, 60
  mov rdi, 0
  syscall
```

Let's compile and run:

```
pc@pc-mytechnotalent:~/Desktop$ nasm -f elf64 -o 1.o 1.asm
pc@pc-mytechnotalent:~/Desktop$ ld 1.o -o 1
pc@pc-mytechnotalent:~/Desktop$ ./1
Hello World!
```

As we can see from last week we successfully created our simple "Hello World" program successfully.

In prior lessons I touched upon the x64 register set however I will review again with this table:

64-bit	32-bit	16-bit	8-bit
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
RIP	EIP	IP	
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

In prior lessons we described what these registers basic functionality consists of however it is important to understand the 64-bit to 8-bit slices of the registers. Registers hold temporary memory. This is the key takeaway here.

We have three sections in Linux-based assembly which consist of a:

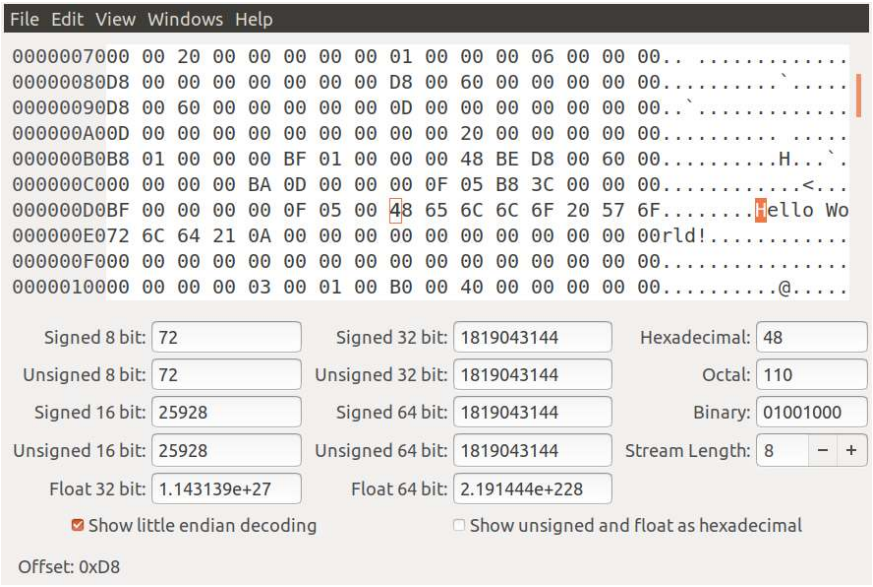
.data = consist of data definitions

.bss = consist of variable data allocation

.text = actual code

In our example above we used the label of **text** not to be confused with the **.text** section. Our compiler will take all of our labels and determine an actual mapped memory location and replace each label with the memory in the actual binary file.

It is important to understand that each string character is a byte in length which is represented by two hex digits. There is an ascii table that you can Google that will show you all of these values. Each hex digit is a nibble or 4-bits long. For example our '**H**' is **0x48** and '**e**' is **0x65**. Let's look at our binary in a hex editor to illustrate.



In last week's lecture's comments, Aaron pointed out something that is very critical that you understand when looking at Assembly in an Operating System vs Firmware such as the code we created for our Operating System in our prior lectures.

Aaron carefully pointed out in the comments last week that a SYSCALL is completely dependent on the operating system. System calls will differ depending on the OS because each OS has a different Kernel and each have their own vector interrupt tables which have an ID associated with them with a corresponding number value.

A SYSCALL is nothing more then when a binary requests a service from a respective kernel to which will take arguments or a list of inputs. It is important to understand in x64 that System Call arguments or inputs correspond to specific registers:

SYSCALL ID	RAX
1	RDI
2	RSI
3	RDX
4	R10
5	R8
6	R9

There are 328 SYSCALLS in a traditional linux kernel. As we see above in our code we use both the **SYS_WRITE** and **SYS_EXIT**. Let's illustrate:

SYSCALL	ID	Argument 1	Argument 2	Argument 3
SYS_WRITE	1	1	text	13
SYS_EXIT	60	0		

Please take a moment to look at our code above to see how this works. In **SYS_WRITE** we load **1** into **RAX** which is our SYSCALL. We load **1** into **RDI** which is our first argument which represents our standard output (0 = standard input & 2 = standard error). Our second argument is loaded into **RSI** which is our **text** label to which when compiled will have an actual memory address as you will see this in a debugger. Finally our third argument will hold the string length which is **13** in our case and loaded into **RDX**. As an exercise I want you to write out how **SYS_EXIT** does the same and keep in mind there is only 1 argument there. PLEASE REVIEW the code above to firmly understand this before moving on!

In addition we have our **_start** label to which our respective operating system will look for otherwise it will throw an error when it seeks to find an entry point to our code. The **global** declaration tells the linker the actual address of the data.

Next week we will debug the binary in GDB.

Part 32 - x64 Assembly [Part 6]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code.

```
section .data
  text db "Hello World!", 0xA

section .text
  global _start

_start:
  mov rax, 1                    ; write system call
  mov rdi, 1
  mov rsi, text
  mov rdx, 13
  syscall

exit:
  mov rax, 60
  mov rdi, 0
  syscall
```

Compile...

```
pc@pc-mytechnotalent:~/Desktop$ nasm -f elf64 -o 1.o 1.asm
pc@pc-mytechnotalent:~/Desktop$ ld 1.o -o 1
pc@pc-mytechnotalent:~/Desktop$ ./1
Hello World!
```

Debug...

```
pc@pc-mytechnotalent:~/Desktop$ gdb -q ./1
Reading symbols from ./1...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x4000b0
(gdb) r
Starting program: /home/pc/Desktop/1

Breakpoint 1, 0x00000000004000b0 in _start ()
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function _start:
=> 0x00000000004000b0 <+0>:     mov    eax,0x1
   0x00000000004000b5 <+5>:     mov    edi,0x1
   0x00000000004000ba <+10>:    movabs rsi,0x6000d8
   0x00000000004000c4 <+20>:    mov    edx,0xd
   0x00000000004000c9 <+25>:    syscall
End of assembler dump.
```

Let's evaluate what is inside the memory address of 0x6000d8.

```
(gdb) x/s 0x6000d8
0x6000d8:       "Hello World!\n"
```

As we can see "**Hello World**" with the return character will then be moved into our **RSI** register.

Next week we will examine this a bit closer.

Part 33 - x64 Assembly [Part 7]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's again review our source code.

```
section .data
  text db "Hello World!", 0xA

section .text
  global _start

_start:
  mov rax, 1                    ; write system call
  mov rdi, 1
  mov rsi, text
  mov rdx, 13
  syscall

exit:
  mov rax, 60
  mov rdi, 0
  syscall
```

Let's compile...

```
pc@pc-mytechnotalent:~/Desktop$ nasm -f elf64 -o 1.o 1.asm
pc@pc-mytechnotalent:~/Desktop$ ld 1.o -o 1
pc@pc-mytechnotalent:~/Desktop$ ./1
Hello World!
```

As we have seen before it produces our string.

```
pc@pc-mytechnotalent:~/Desktop$ gdb -q ./1
Reading symbols from ./1...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x4000b0
(gdb) r
Starting program: /home/pc/Desktop/1

Breakpoint 1, 0x00000000004000b0 in _start ()
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function _start:
=> 0x00000000004000b0 <+0>:     mov    eax,0x1
   0x00000000004000b5 <+5>:     mov    edi,0x1
   0x00000000004000ba <+10>:    movabs rsi,0x6000d8
   0x00000000004000c4 <+20>:    mov    edx,0xd
   0x00000000004000c9 <+25>:    syscall
End of assembler dump.
```

We debug and see the string being moved into **0x6000d8** and then **RSI**.

```
(gdb) x/s 0x6000d8
0x6000d8:       "Hello World!\n"
```

Just to verify we can see the string at the aforementioned address. NOW FOR A BIT OF FUN :)...

```
(gdb) set {char [15]} 0x6000d8 = "Hacked World!\n"
```

Here we demonstrate we have the power to simply hack and redefine the string in memory. We are simply setting a char byte length and setting a new string.

```
0x6000d8:       "Hacked World!\n"
```

As we can see we have successfully altered the string in memory.

```
(gdb) si
0x00000000004000b5 in _start ()
(gdb) si
0x00000000004000ba in _start ()
(gdb) si
0x00000000004000c4 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00000000004000b0 <+0>:     mov    eax,0x1
   0x00000000004000b5 <+5>:     mov    edi,0x1
   0x00000000004000ba <+10>:    movabs rsi,0x6000d8
=> 0x00000000004000c4 <+20>:    mov    edx,0xd
   0x00000000004000c9 <+25>:    syscall
End of assembler dump.
(gdb) x/s $rsi
0x6000d8:       "Hacked World!\n"
```

We continue and run through the binary and see that our hack continues through **RSI**.

```
(gdb) c
Continuing.
Hacked World![Inferior 1 (process 23979) exited normally]
```

Finally we see when we run the binary we have successfully hacked it's operation. This is a very simple example however shows the power of truly understanding assembly at this level. GUI debugger tools will also provide this functionality however I like to use the command line tools so that they could be used on every environment.

The purpose of these tools is to UNDERSTAND how this is done and what to look for when you are professionally reversing in real-time. You need to understand how an attacker can alter memory and/or instructions. We need more professional RE's to help defend infrastructures throughout the world and hopefully these tutorials motivate you toward a career in such.

Part 34 - x64 C++ 1 Code [Part 1]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Today we start our RE with the C++ language. The vast majority of malware is written in C++ and walking through simple code examples over the coming months and breaking them down in a debugger will give you a real hands-on approach to learning true RE.

We will use Kali Linux going forward with Radare 2. You can get VirtualBox and download the Kali Linux x64 Appliance to follow along.

Let's start with the C++ 1 code example:

```
1 #include <iostream>
2
3 int main(int argc, char** argv) {
4   std::cout << "Hello World\n";
5
6   return 0;
7 }
```

Here we simply create a main function and use the C++ output stream library to output the text "Hello World" with a new line at the end to the terminal. Let's compile and link:

```
root@kali:~/Documents# g++ 1.cpp -o 1
```

Let's run in the terminal:

```
root@kali:~/Documents# ./1
Hello World
```

As we can see "Hello World" successfully echoed to the terminal.

Next week we will introduce Radare 2 and debug the code and examine what it looks like in x64 Assembly.

Part 35 - x64 C++ 2 Debug [Part 2]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code:

```
#include <iostream>

int main(int argc, char** argv) {
  std::cout << "Hello World\n";

  return 0;
}
```

Compile:

```
root@kali:~/Documents# g++ 1.cpp -o 1
```

Run:

```
root@kali:~/Documents# ./1
Hello World
```

For literally years I have been using GDB as the debugger of choice. The reason is that it is on every Linux based system which runs just about every IoT and Server in the world. In addition, there are versions for Windows.

I have struggled hard with this but have decided to introduce another terminal based debugger called Radare 2. The reason I like Radare 2 so much is that it is still terminal based yet more robust with its feature set. If you are running a Kali Linux VM like I am here you can simply the below.

Let's open up our binary for write mode and simply analyze the binary.

```
root@kali:~/Documents# r2 -w ./1
[0x00001070]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00001070]> s sym.main
[0x00001155]> pdf
            ;-- main:
/ (fcn) sym.main 41
|   sym.main (int argc, char **argv, char **envp);
|           ; var char **local_10h @ rbp-0x10
|           ; var int local_4h @ rbp-0x4
|           ; arg int argc @ rdi
|           ; arg char **argv @ rsi
|           ; DATA XREF from entry0 (0x108d)
|           0x00001155      55              push rbp
|           0x00001156      4889e5          mov rbp, rsp
|           0x00001159      4883ec10        sub rsp, 0x10
|           0x0000115d      897dfc          mov dword [local_4h], edi   ; argc
|           0x00001160      488975f0        mov qword [local_10h], rsi  ; argv
|           0x00001164      488d359a0e00.   lea rsi, qword str.Hello_World ; 0x2005 ; "Hello World\n"
|           0x0000116b      488d3dce2e00.   lea rdi, qword obj.__TMC_END ; loc.__bss_start ; 0x4040
|           0x00001172      e8c9feffff      call sym.std::basic_ostream_char_std::char_traits_char___
|           0x00001177      b800000000      mov eax, 0
|           0x0000117c      c9              leave
\           0x0000117d      c3              ret
```

Ok, there is a lot going on here. Let's break it down. First, we open up Radare 2 in write mode by typing '**r2 -w ./1**' and then use the '**aaa**' command to analyze the binary. We then use '**s sym.main**' to seek to the main routine of the binary which is our entry point. We then do a '**pdf**' command to disassemble the binary.

We see what we refer to as the prologue where we push **rbp** the stack base pointer onto the stack. We then move **rsp** into **rbp** for safe keeping and then we reserve **0x10** hex bytes or 16 decimal bytes on the stack to make room for our string.

If none of this makes sense please go back to the beginning of the tutorial series to review basic assembly and the registers as it is CRITICAL you understand this before we move forward.

We can clearly see the qword of '**Hello World\n**' at memory address **0x2005** and then we see our C++ library call for the output stream which is **cout** to display our string to the terminal.

Let's examine **0x2005** to verify that our string is at that location:

```
[0x00001155]> psz @ 0x2005
Hello World
```

Next week we will hack the value and modify the binary. I highly encourage you all to install VirtualBox which is free and get the Kali Linux VirtualBox image and install Vim as well.

There are tutorials on all of this in my prior series. Stay tuned for the hack next week!

Part 36 - x64 C++ 3 Hacking [Part 3]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's review our code:

```
#include <iostream>

int main(int argc, char** argv) {
  std::cout << "Hello World\n";

  return 0;
}
```

Compile:

```
root@kali:~/Documents# g++ 1.cpp -o 1
```

Run:

```
root@kali:~/Documents# ./1
Hello World
```

Let's remember this line above when we compare against our hacked binary.

Let's open up our binary for write mode and simply analyze the binary.

```
root@kali:~/Documents# r2 -w ./1
[0x00001070]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00001070]> s sym.main
[0x00001155]> pdf
            ;-- main:
/ (fcn) sym.main 41
|   sym.main (int argc, char **argv, char **envp);
|           ; var char **local_10h @ rbp-0x10
|           ; var int local_4h @ rbp-0x4
|           ; arg int argc @ rdi
|           ; arg char **argv @ rsi
|           ; DATA XREF from entry0 (0x108d)
|           0x00001155      55              push rbp
|           0x00001156      4889e5          mov rbp, rsp
|           0x00001159      4883ec10        sub rsp, 0x10
|           0x0000115d      897dfc          mov dword [local_4h], edi   ; argc
|           0x00001160      488975f0        mov qword [local_10h], rsi  ; argv
|           0x00001164      488d359a0e00.   lea rsi, qword str.Hello_World ; 0x2005 ; "Hello World\n"
|           0x0000116b      488d3dce2e00.   lea rdi, qword obj.__TMC_END ; loc.__bss_start ; 0x4040
|           0x00001172      e8c9feffff      call sym.std::basic_ostream_char_std::char_traits_char___
|           0x00001177      b800000000      mov eax, 0
|           0x0000117c      c9              leave
\           0x0000117d      c3              ret
```

Ok, there is a lot going on here. Let's break it down. First, we open up Radare 2 in write mode by typing '**r2 -w ./1**' and then use the '**aaa**' command to analyze the binary. We then use '**s sym.main**' to seek to the main routine of the binary which is our entry point. We then do a '**pdf**' command to disassemble the binary.

We see what we refer to as the prologue where we push **rbp** the stack base pointer onto the stack. We then move **rsp** into **rbp** for safe keeping and then we reserve **0x10** hex bytes or 16 decimal bytes on the stack to make room for our string.

If none of this makes sense please go back to the beginning of the tutorial series to review basic assembly and the registers as it is CRITICAL you understand this before we move forward.

We can clearly see the qword of '**Hello World\n**' at memory address **0x2005** and then we see our C++ library call for the output stream which is **cout** to display our string to the terminal.

Let's examine **0x2005** to verify that our string is at that location:

```
[0x00001155]> psz @ 0x2005
Hello World
```

NOW TIME FOR THE HACK!

Let's hack the value to something like:

```
[0x00001155]> w Hacked World\n @ 0x2005
```

Now let's see what is now inside memory value @ **0x2005**!

```
[0x00001155]> psz @ 0x2005
Hacked World
```

BOOM! As we can see we have hacked the value and when we quit Radare 2 it will write it and modify our binary as such.,

```
[0x00001155]> q
root@kali:~/Documents/Projects# ./1
Hacked World
```

As you can see we have hacked the binary! This is very basic but now you have an elementary level of understanding of Reverse Engineering a C++ binary.

Next week we will continue our journey into C and step-by-step reverse engineering.

Part 37 - x64 C & Genesis Of Life

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Congrats you wrote, compiled and hacked your first C++ program. For the rest of this tutorial I am going to focus on the father of all programming languages from "Hello World" to web servers in the programming language to which ALL modern languages come from C.

Like the variety of religions there are programming languages. Nonetheless there is the ROOT religion or language to which all spawn which is C. I am going to over the next several months teach you C and Reverse Engineer each binary so you have a mastery over the MASTER language of all existence.

When we need to develop in an agile environment we will of course use Java or Python or any of the other rapid development languages however if you are to master Cyber Engineering you MUST become ONE with the WORD to which in digital and cyber terms is the C Programming Language.

Think of C as if you are in church where Python or Java or C# you are in a secular environment. C will allow TOTAL and complete control over your program or environment where Java or Python will allow only partial control however they are NECESSARY languages in today's rapid development business logic environments.

In our next lesson we begin with the basic "hello world" program as we did in our prior lesson however we now will work with C. Remember Einstein - "**I want to know God's thoughts, the rest are details.**" This is the difference between C and any other language you are sitting at the ROOT of engineering design for portable systems!

Part 38 - x64 Networking Basics

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Ok so what now? Where are we in the world? What is our purpose? What shall I focus on? What shall I learn?

There are over 30 billion devices connected to the Internet today. Nonetheless, the common thread in all basic architecture is the C programming language.

We have established that networking can be described in a very high-level pseudo framework called the OSI Model which has 7 layers.

PLEASE **D**O **N**OT **T**HROW **S**AUSAGE **P**IZZA **A**WAY. Ok I am not insane, well, ok I am but this is a good standard agreed upon way to remember the layers in the OSI model which is our Open Systems Interconnection model.

1)**PHYSICAL LAYER** - Raw electrical layer which read voltages on an ethernet cable or reading the Wi-Fi RF (radio frequencies). Protocols associated: USB, DSL, ISDN, Infrared, etc...

2)**DATA LINK LAYER** - Deals with how a message between notes starts and ends called framing which has some error correction, detection and some flow control. Protocols associated: Ethernet, VLAN, etc...

3)**NETWORK LAYER** - Transmits packets between nodes in different networks which involves routing. Protocols associated: IPX, NAT, ICMP, ARP, etc...

4)**TRANSPORT LAYER** - Reliably deliver data between two hosts which must split it up into chunks to send. Protocols associated: NetBIOS, TCP, UDP, etc...

5)**SESSION LAYER** - Adds checkpoint and resume in addition to term dialogues. Protocols associated: SMB, SOCKS, etc...

6)**PRESENTATION LAYER** - Where data structure for and presentation for an application are created where we have encoding, serialization and encryption. Protocols associated: TLS, SSL, etc...

7)**APPLICATION LAYER** - Web browsers and apps that use web interfaces like email, etc. Protocols associated: DHCP, DNS, HTTP, HTTPS, POP3, SMTP, FTP, TELNET, etc...

As we browse a website we start at the PHYSICAL and go to the APP and as it hits the server it is at the APP and goes back down to the PHYSICAL and back through the cycle.

This is an important series of concepts that you must understand in any basic networking. This is NOT a course in networking as we will touch BRIEFLY on these concepts so I would suggest you find a free course on YouTube for networking if you are stuck. I want to get through some basic theory so we can work with C networking apps.

Part 39 - Why C?

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

So... What does an x64 server or computer actually understand? 0100010100100100101010 and many more...

A small level above that we are at machine code which is a series of hex digits which translate into machine instructions and/or data.

With the C programming language, we created a construct to more easily create programs to communicate with the hardware. C is the Grandfather of almost every programming language in modern existence.

C abstracts away the x64 binary of 010101000101001011 or machine code of 0x90 0x45 0x22 0x22 or assembly mov rax, 0x222323123, etc...

Next we create our first real C program!

Part 40 - Hacking Hello World!

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Ok it is time we look at the most basic C program, debug it and hack it. If we are to have mastery we must create and destroy in a single-step so that we have mastery over the domain.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Hello World!\n");
6   return 0;
7 }
```

Let us fire up VIM and type out the following. We include our standard library and create a main function to which we use the library function of printf to echo a string of chars and since the type of main is int meaning integer we return 0.

Let us compile and see what happens when we run:

```
root@kali:~/Documents/Projects# gcc 1.c -o 1
root@kali:~/Documents/Projects# ./1
Hello World!
```

As we see like we did in our C++ example we see '**Hello World!**' echoed successfully.

Let's debug in Radare:

```
root@kali:~/Documents/Projects# r2 -w ./1
[0x00001050]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
```

This is simple, we use **aaa** to analyze the binary and seek to main with **s sym.main**.

Let's look at the assembly and analyze:

```
[0x00001050]> s sym.main
[0x00001135]> pdf
            ;-- main:
/ (fcn) sym.main 23
|   sym.main (int argc, char **argv, char **envp);
|           ; DATA XREF from entry0 (0x106d)
|           0x00001135      55              push rbp
|           0x00001136      4889e5          mov rbp, rsp
|           0x00001139      488d3dc40e00.   lea rdi, qword str.Hello_World ; 0x20
04 ; "Hello World!" ; const char *s
|           0x00001140      e8ebfeffff      call sym.imp.puts           ; int put
s(const char *s)
|           0x00001145      b800000000      mov eax, 0
|           0x0000114a      5d              pop rbp
\           0x0000114b      c3              ret
```

Assembly! The definition of raw sexy!

I went over this in detail in the previous lessons on Assembly but let us review.

1)We **push rbp** which means we push the value currently in the base pointer onto the stack.

2)We **lea rdi, qword str.Hello_World** which means we load the effective address of the quad word of our string into the **rdi** register. So far should be simple for you to follow along.

3)We then **call sym.imp.puts** um wait! We used **printf** what the hell! Well our compiler optimizes our code and the compiler chose the **puts** function in the stdio library to echo the string to our terminal. Again easy enough.

4)We clean out **eax** and then pop the original value in the **rbp** register back into **rbp**. If you are confused by this review the earlier part of the series please.

We know our string '**Hello World!**' lives at a pretty house in Arlington, VA at the address of **0x2004** well ok, it's not Arlington, VA but it is in mapped memory (since we are not technically debugging we are messing with mapped code meaning the same values on disk).

```
[0x00001135]> psz @ 0x2004
Hello World!
```

To confirm we see the value at **0x2004** is '**Hello World!**' Let's hack that value to anything we want with the **w** command and write directly to that mapped memory address.

```
[0x00001135]> w Hacked World! @ 0x2004
```

Let us re-examine who NOW lives in our Arlington, VA house!

```
[0x00001135]> psz @ 0x2004
Hacked World!
```

Success! We hacked the value and when we exit our debugger we see:

```
[0x00001135]> q
root@kali:~/Documents/Projects# ./1
Hacked World!
```

We have successfully altered the binary.

This is alot to digest here. If you are stumped ask questions in the comments PLEASE! Do not continue as I am here to help. It is CRITICAL you understand these most basic things before we continue!

Part 41 - Hacking Variables!

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

In C we have several data types to which we can create variables. I will use a few simple examples:

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5   char a = 'a';
 6   int b = 1;
 7   double c = 1.1;
 8
 9   printf("char a = %c\n", a);
10   printf("int b = %d\n", b);
11   printf("double c = %f\n", c);
12
13   return 0;
14 }
```

Let's compile and run:

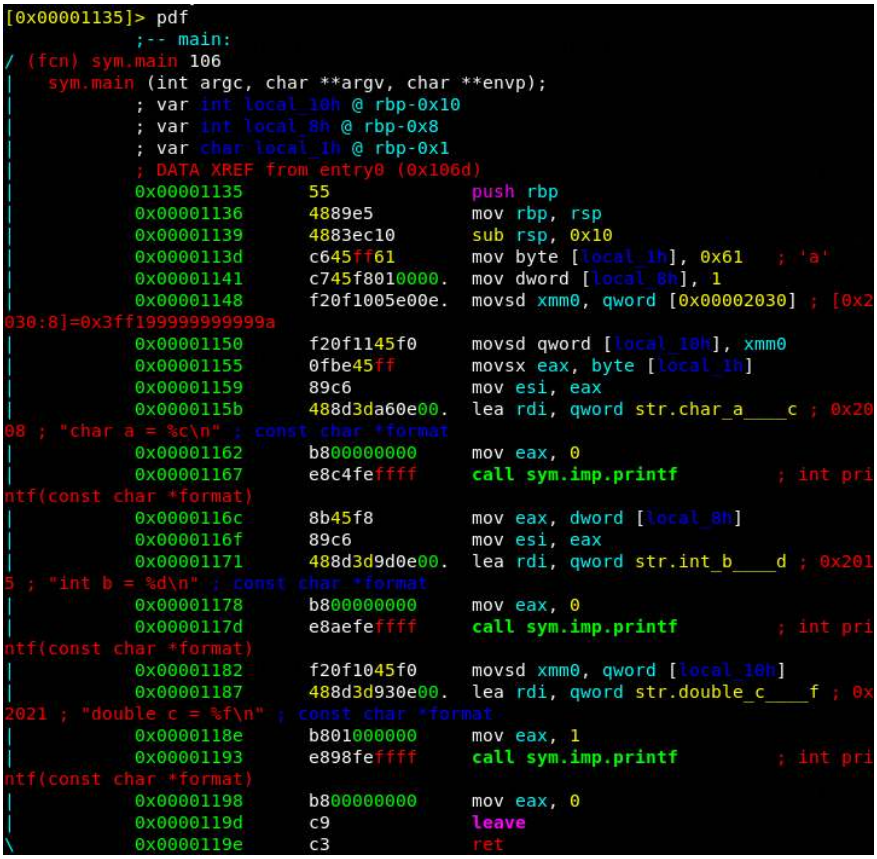
```
root@kali:~/Documents/Projects# gcc 2.c -o 2
root@kali:~/Documents/Projects# ./2
char a = a
int b = 1
double c = 1.100000
```

Ok as we can see we have a character an integer and a double. These are some of the most basic data types in C to which we have created a series of variables as shown above.

Let us load the binary into Radare:

```
root@kali:~/Documents/Projects# r2 -w ./2
[0x00001050]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00001050]> s sym.main
```

Let's disassemble at main:



```
[0x00001135]> pdf
            ;-- main:
/ (fcn) sym.main 106
|   sym.main (int argc, char **argv, char **envp);
|           ; var int local_10h @ rbp-0x10
|           ; var int local_8h @ rbp-0x8
|           ; var char local_1h @ rbp-0x1
|           ; DATA XREF from entry0 (0x106d)
|           0x00001135      55              push rbp
|           0x00001136      4889e5          mov rbp, rsp
|           0x00001139      4883ec10        sub rsp, 0x10
|           0x0000113d      c645ff61        mov byte [local_1h], 0x61   ; 'a'
|           0x00001141      c745f8010000.   mov dword [local_8h], 1
|           0x00001148      f20f1005e00e.   movsd xmm0, qword [0x00002030] ; [0x2
030:8]=0x3ff1999999999999a
|           0x00001150      f20f1145f0      movsd qword [local_10h], xmm0
|           0x00001155      0fbe45ff        movsx eax, byte [local_1h]
|           0x00001159      89c6            mov esi, eax
|           0x0000115b      488d3da60e00.   lea rdi, qword str.char_a____c ; 0x20
08 ; "char a = %c\n" ; const char *format
|           0x00001162      b800000000      mov eax, 0
|           0x00001167      e8c4feffff      call sym.imp.printf         ; int pri
ntf(const char *format)
|           0x0000116c      8b45f8          mov eax, dword [local_8h]
|           0x0000116f      89c6            mov esi, eax
|           0x00001171      488d3d9d0e00.   lea rdi, qword str.int_b____d ; 0x201
5 ; "int b = %d\n" ; const char *format
|           0x00001178      b800000000      mov eax, 0
|           0x0000117d      e8aefeffff      call sym.imp.printf         ; int pri
ntf(const char *format)
|           0x00001182      f20f1045f0      movsd xmm0, qword [local_10h]
|           0x00001187      488d3d930e00.   lea rdi, qword str.double_c____f ; 0x
2021 ; "double c = %f\n" ; const char *format
|           0x0000118e      b801000000      mov eax, 1
|           0x00001193      e898feffff      call sym.imp.printf         ; int pri
ntf(const char *format)
|           0x00001198      b800000000      mov eax, 0
|           0x0000119d      c9              leave
\           0x0000119e      c3              ret
```

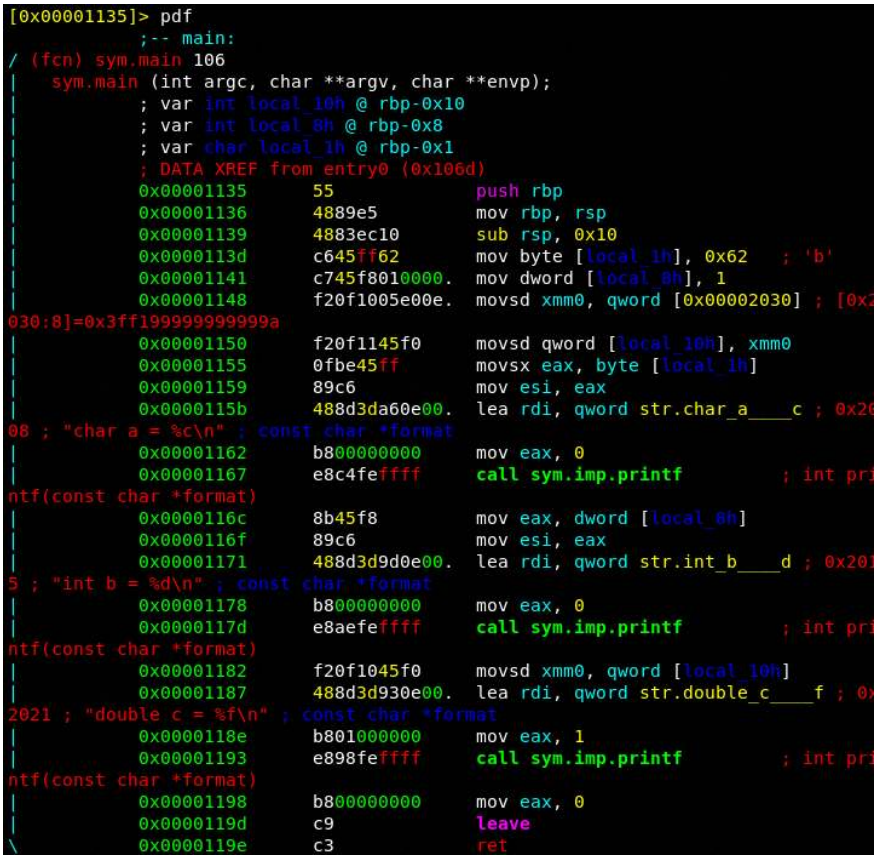
Ok very simply we see 3 variable declarations defined up at the top in reverse order as they are **local_1h** which is our **char a, local_8h** which is our **int b** and **local_10h** which is our **double c**. You can also see the **rbp** base pointer allocating space for these variables. This is nice pseudo code that the debugger shows you up top.

Ok stay with me!

Within memory at **0x0000113d** we see the instructions **mov byte [local1_h], 0x61** which is in our ascii table a lowercase '**a**'. We know that [local1_h] is not real code however what is going on under the hood is the fact that these variables are pushed onto the stack in reverse order as we can see above. Therefore, if we were to hack our code to something like **mov byte [rbp-0x1], 0x62** what do you think might happen? Very simple, we know that in reality the code at the mapped memory address of **0x0000113d** what is really going on is **mov byte [rbp-0x1], 0x61**. Quite simply what we have just done is hack our value of '**a**' to '**b**'. This should hopefully make sense to you.

```
[0x00001135]> wa mov byte [rbp-0x1], 0x62 @ 0x0000113d
Written 4 byte(s) (mov byte [rbp-0x1], 0x62) = wx c645ff62
```

Now let us re-examine our binary:



As we can clearly see at memory address **0x0000113d** we in fact see '**b**'. We have successfully hacked this portion.

```
root@kali:~/Documents/Projects# ./2
char a = b
int b = 1
double c = 1.100000
```

We exit out of Radare and re-run the binary and we can see we have successfully hacked the value.

HOMEWORK TIME! I want you to with this knowledge now hack the **int** and the **double**. I want you to put your results in the comment sections below. It is VERY important that you type all of this out and actually explore the exercises so I am looking forward to seeing your hacks in the comments!

Part 42 - Hacking Branches!

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

Let's take a look at some branching logic:

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5   int a = 1;
 6
 7   if (a == 1)
 8   {
 9     printf("A is 1!\n");
10   }
11   else
12   {
13     printf("A is NOT 1!\n");
14   }
15
16   return 0;
17 }
```

As we can plainly see we init an int to 1 and if the variable is equal to 1 the first if statement prints a response to standard output.

Let's compile:

```
root@kali:~/Documents/Projects# gcc 3.c -o 3
```

Let's run:

```
root@kali:~/Documents/Projects# ./3
A is 1!
```

As we can logically see the first branch is taken. Let's take it into Radare and look around at the assembly:

```
root@kali:~/Documents/Projects# r2 -w ./3
[0x00001050]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00001050]> s sym.main
[0x00001135]> pdf
/ (fcn) main 54
|   main (int argc, char **argv, char **envp);
|           ; var unsigned int local_4h @ rbp-0x4
|           ; DATA XREF from entry0 (0x106d)
|           0x00001135      55              push rbp
|           0x00001136      4889e5          mov rbp, rsp
|           0x00001139      4883ec10        sub rsp, 0x10
|           0x0000113d      c745fc010000.   mov dword [local_4h], 1
|           0x00001144      837dfc01        cmp dword [local_4h], 1
|       ,=< 0x00001148      750e            jne 0x1158
|       |   0x0000114a      488d3db30e00.   lea rdi, qword str.A_is_1   ; 0x2004
; "A is 1!" ; const char *s
|       |   0x00001151      e8dafeffff      call sym.imp.puts           ; int put
s(const char *s)
|      ,==< 0x00001156      eb0c            jmp 0x1164
|      ||   ; CODE XREF from main (0x1148)
|      |`-> 0x00001158      488d3dad0e00.   lea rdi, qword str.A_is_NOT_1 ; 0x200
c ; "A is NOT 1!" ; const char *s
|      |    0x0000115f      e8ccfeffff      call sym.imp.puts           ; int put
s(const char *s)
|      |    ; CODE XREF from main (0x1156)
|      `--> 0x00001164      b800000000      mov eax, 0
|           0x00001169      c9              leave
\           0x0000116a      c3              ret
```

We can see the branching logic with the aqua colored arrows. At **0x0000114a** we see our first branch being loaded into **rdi**. Take note at **0x00001148** we see a **jne 0x1158**. At **0x00001158** we see our second branch being loaded into **rdi**.

The **jne** means jump if not equal. This means if what is being compared in **0x00001144** is not equal to 1 (we see **1** being compared to what is in **local_4h** which we know is pseudo code for what is actually in **rbp-0x4**. This should make sense as I went over this in detail last week if you are confused please revisit our last lesson.

To hack we simply make the **jne** statement to **je** which is jump if equal which we know the **cmp** or comparison is equal so it will now branch to "**A is NOT 1!**".

```
[0x00001135]> wa je 0x1158 @ 0x00001148
Written 2 byte(s) (je 0x1158) = wx 740e
```

When we exit Radare we can see we have hacked the binary successfully:

```
root@kali:~/Documents/Projects# ./3
A is NOT 1!
```

Stay tuned!

Part 43 - Hacking Pointers!

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. https://github.com/mytechnotalent/Reverse-Engineering-Tutorial

We are at the end of the road. This is the final video in the x64 series. The final topic is that of pointers.

What are pointers? Let us start with an example.

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5   int lottery_number = 777;
 6
 7   printf("Address \t Variable Name \t\t Value \n");
 8   printf("%p \t %s \t %d \n", &lottery_number, "Lottery Number", lottery_num
   ber);
 9
10   return 0;
11 }
```

A pointer is nothing more than a memory address. When we compile we will clearly see where lottery_number lives in mapped memory (this is a running example unlike our unmapped Radare examples).

```
root@kali:~/Documents/Projects# gcc 4.c -o 4
root@kali:~/Documents/Projects# ./4
Address          Variable Name          Value
0x7ffc7f690c6c   Lottery Number         777
```

Let's add a true pointer to the example:

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5   int lottery_number = 777;
 6
 7   printf("Address \t Variable Name \t\t Value \n");
 8   printf("%p \t %s \t %d \n", &lottery_number, "Lottery Number", lottery_num
   ber);
 9
10   int * p_lottery_number = &lottery_number;
11
12   printf("%p \t %s \t %d \n", p_lottery_number, "Lottery Number", lottery_nu
   mber);
13
14   return 0;
15 }
```

We see the same value:

```
root@kali:~/Documents/Projects# ./4
Address          Variable Name          Value
0x7ffcdb5603a4   Lottery Number         777
0x7ffcdb5603a4   Lottery Number         777
```

Let us experiment more:

```c
#include <stdio.h>

int main(void)
{
  int lottery_number = 777;

  printf("Address \t Variable Name \t\t Value \n");
  printf("%p \t %s \t %d \n", &lottery_number, "Lottery Number", lottery_number);

  int * p_lottery_number = &lottery_number;

  printf("%p \t %s \t %d \n", p_lottery_number, "Lottery Number", lottery_number);

printf("%p \t %s \t %d \n", &p_lottery_number, "Lottery Number", *p_lottery_number);


  return 0;
}
```

We see the pointer address point to a new address:

```
root@kali:~/Documents/Projects# gcc 4.c -o 4
root@kali:~/Documents/Projects# ./4
Address          Variable Name           Value
0x7ffe1aa029ec   Lottery Number          777
0x7ffe1aa029ec   Lottery Number          777
0x7ffe1aa029e0   Lottery Number          777
```

Remember pointers are memory addresses of other variables. Let's look at it another way:

```c
#include <stdio.h>

int main(void)
{
  int lottery_number = 777;
  int * p_lottery_number = &lottery_number;

  printf("Address \t Variable Name \t\t Value \n");
  printf("%p \t %s \t %d \n", p_lottery_number, "Lottery Number", lottery_number);
  printf("%p \t %s \t %d \n", &p_lottery_number, "Lottery Number", p_lottery_number);

  return 0;
}
```

Let us compile:

```
root@kali:~/Documents/Projects# gcc 4.c -o 4
root@kali:~/Documents/Projects# vim 4.c
root@kali:~/Documents/Projects# ./4
Address          Variable Name           Value
0x7ffe01b8815c   Lottery Number          777
0x7ffe01b88150   Lottery Number          28868956
```

We deference by doing the following:

```c
#include <stdio.h>

int main(void)
{
  int lottery_number = 777;
  int * p_lottery_number = &lottery_number;

  printf("Address \t Variable Name \t\t Value \n");
  printf("%p \t %s \t %d \n", p_lottery_number, "Lottery Number", lottery_number);
  printf("%p \t %s \t %d \n", &p_lottery_number, "Lottery Number", p_lottery_number);

  printf("\n*p_lottery_number: %d \n", *p_lottery_number);

  return 0;
}
```

Then we compile:

```
root@kali:~/Documents/Projects# gcc 4.c -o 4
root@kali:~/Documents/Projects# ./4
Address          Variable Name          Value
0x7ffe908d805c   Lottery Number         777
0x7ffe908d8050   Lottery Number         -1869774756

*p_lottery_number: 777
```

We can see the deference pointer is equal to 777.

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5   int lottery_number;
 6   int winning_numbers[3] = {4, 2, 3};
 7
 8   printf("Element \t Address \t Value \n");
 9
10   for(lottery_number = 0; lottery_number < 3; lottery_number++)
11   {
12     printf("winning_numbers[%d] \t %p \t %d \n", lottery_number, &wi
   nning_numbers[lottery_number], winning_numbers[lottery_number]);
13   }
14
15
16   return 0;
17 }
```

We can see the example with an array:

```
root@kali:~/Documents/Projects# gcc 4.c -o 4
root@kali:~/Documents/Projects# ./4
Element          Address          Value
winning_numbers[0]       0x7ffe5d48b3d0          4
winning_numbers[1]       0x7ffe5d48b3d4          2
winning_numbers[2]       0x7ffe5d48b3d8          3
```

Let's debug:

```
root@kali:~/Documents/Projects# radare2 -w ./4
[0x00001060]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[ ] Constructing a function name for fcn.* and sym.func.* functions (aan[x] Cons
tructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00001060]> s sym.main
[0x00001145]> pdf
```

Then we disassemble:

```
/ (fcn) main 121
|   main (int argc, char **argv, char **envp);
|           ; var int local_10h @ rbp-0x10
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var signed int local_4h @ rbp-0x4
|           ; arg void *arg_10h @ rbp+0x10
|           ; DATA XREF from entry0 (0x107d)
|           0x00001145      55              push rbp
|           0x00001146      4889e5          mov rbp, rsp
|           0x00001149      4883ec10        sub rsp, 0x10
|           0x0000114d      c745f0040000.   mov dword [local_10h], 4
|           0x00001154      c745f4020000.   mov dword [local_ch], 2
|           0x0000115b      c745f8030000.   mov dword [local_8h], 3
|           0x00001162      488d3d9f0e00.   lea rdi, qword str.Element___Address
___Value ; 0x2008 ; "Element \t Address \t Value " ; const char *s
|           0x00001169      e8c2feffff      call sym.imp.puts           ; int put
s(const char *s)
|           0x0000116e      c745fc000000.   mov dword [local_4h], 0
|       ,=< 0x00001175      eb3a            jmp 0x11b1
|       |   ; CODE XREF from main (0x11b5)
|      .--> 0x00001177      8b45fc          mov eax, dword [local_4h]
|      :|   0x0000117a      4898            cdqe
|      :|   0x0000117c      8b5485f0        mov edx, dword [rbp + rax*4 - 0x10]
|      :|   0x00001180      488d45f0        lea rax, qword [local_10h]
|      :|   0x00001184      8b4dfc          mov ecx, dword [local_4h]
|      :|   0x00001187      4863c9          movsxd rcx, ecx
|      :|   0x0000118a      48c1e102        shl rcx, 2
|      :|   0x0000118e      488d3408        lea rsi, qword [rax + rcx]
|      :|   0x00001192      8b45fc          mov eax, dword [local_4h]
|      :|   0x00001195      89d1            mov ecx, edx
|      :|   0x00001197      4889f2          mov rdx, rsi
|      :|   0x0000119a      89c6            mov esi, eax
|      :|   0x0000119c      488d3d850e00.   lea rdi, qword str.winning_numbers__d
____p____d ; 0x2028 ; "winning_numbers[%d] \t %p \t %d \n" ; const char *form
at
|      :|   0x000011a3      b800000000      mov eax, 0
|      :|   0x000011a8      e893feffff      call sym.imp.printf         ; int pri
ntf(const char *format)
|      :|   0x000011ad      8345fc01        add dword [local_4h], 1
|      :|   ; CODE XREF from main (0x1175)
|      :`-> 0x000011b1      837dfc02        cmp dword [local_4h], 2
```

Let's hack!

```
[0x00001145]> wa mov dword [rbp-0x8], 0x6 @ 0x0000115b
Written 7 byte(s) (mov dword [rbp-0x8], 0x6) = wx c745f806000000
```

Let's re-examine the binary:

```
[0x00001145]> pdf
/ (fcn) main 121
|   main (int argc, char **argv, char **envp);
|           ; var int local_10h @ rbp-0x10
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var signed int local_4h @ rbp-0x4
|           ; arg void *arg_10h @ rbp+0x10
|           ; DATA XREF from entry0 (0x107d)
|           0x00001145      55              push rbp
|           0x00001146      4889e5          mov rbp, rsp
|           0x00001149      4883ec10        sub rsp, 0x10
|           0x0000114d      c745f0040000.   mov dword [local_10h], 4
|           0x00001154      c745f4020000.   mov dword [local_ch], 2
|           0x0000115b      c745f8060000.   mov dword [local_8h], 6
|           0x00001162      488d3d9f0e00.   lea rdi, qword str.Element___Address
___Value ; 0x2008 ; "Element \t Address \t Value " ; const char *s
|           0x00001169      e8c2feffff      call sym.imp.puts           ; int put
s(const char *s)
|           0x0000116e      c745fc000000.   mov dword [local_4h], 0
|       ,=< 0x00001175      eb3a            jmp 0x11b1
|       |   ; CODE XREF from main (0x11b5)
|       .--> 0x00001177      8b45fc          mov eax, dword [local_4h]
|       :|   0x0000117a      4898            cdqe
|       :|   0x0000117c      8b5485f0        mov edx, dword [rbp + rax*4 - 0x10]
|       :|   0x00001180      488d45f0        lea rax, qword [local_10h]
|       :|   0x00001184      8b4dfc          mov ecx, dword [local_4h]
|       :|   0x00001187      4863c9          movsxd rcx, ecx
|       :|   0x0000118a      48c1e102        shl rcx, 2
|       :|   0x0000118e      488d3408        lea rsi, qword [rax + rcx]
|       :|   0x00001192      8b45fc          mov eax, dword [local_4h]
|       :|   0x00001195      89d1            mov ecx, edx
|       :|   0x00001197      4889f2          mov rdx, rsi
|       :|   0x0000119a      89c6            mov esi, eax
|       :|   0x0000119c      488d3d850e00.   lea rdi, qword str.winning_numbers__d
_____p____d ; 0x2028 ; "winning_numbers[%d] \t %p \t %d \n" ; const char *form
at
|       :|   0x000011a3      b800000000      mov eax, 0
|       :|   0x000011a8      e893feffff      call sym.imp.printf         ; int pri
ntf(const char *format)
|       :|   0x000011ad      8345fc01        add dword [local_4h], 1
|       :|   ; CODE XREF from main (0x1175)
|       :`-> 0x000011b1      837dfc02        cmp dword [local_4h], 2
|       `==< 0x000011b5      7ec0            jle 0x1177
|           0x000011b7      b800000000      mov eax, 0
|           0x000011bc      c9              leave
\           0x000011bd      c3              ret
```

We can see we hacked the value of 3 with 6.

```
root@kali:~/Documents/Projects# ./4
Element          Address         Value
winning_numbers[0]       0x7ffdf6600fc0          4
winning_numbers[1]       0x7ffdf6600fc4          2
winning_numbers[2]       0x7ffdf6600fc8          6
```

We can see we have made the successful hack.

I hope over the years through the literal hundreds of x86, ARM and x64 tutorials you have a basic knowledge of how to do GOOD to protect critical infrastructures from malicious hands by understanding how the enemy works. Go and do GOOD work!

The 64-bit ARM Architecture

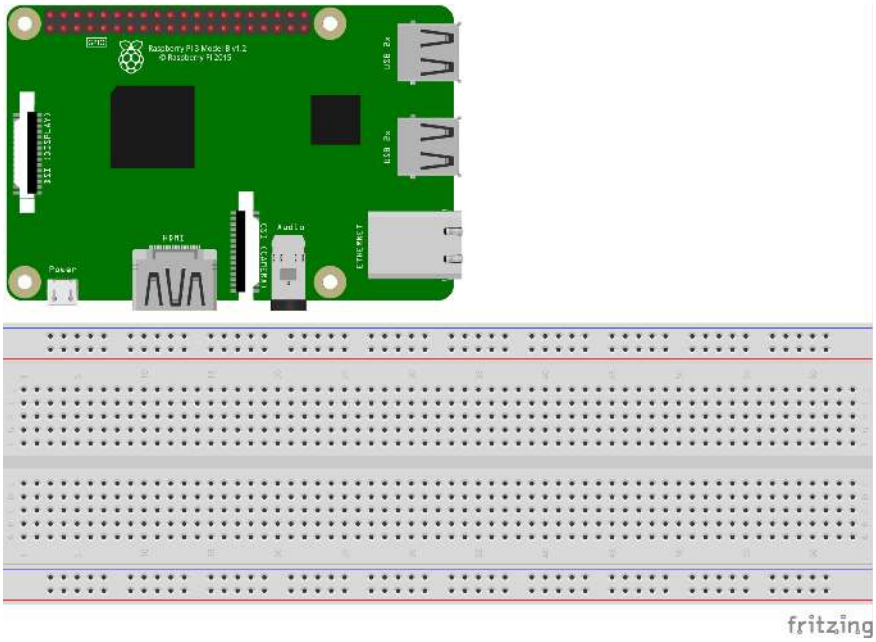
Let's dive in rightaway!

Part 2 - Development Setup

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to set up our development environment. We will need the following:

```
Raspberry Pi 4
64GB MicroSD Card
Micro SD Card Reader/Writer
Download 64-bit Kali Linux ARM Image
Download balenaEtcher
Flash Kali Linux ARM Image
OPTIONAL: Video [Load Kali RPI 4]
How To Install VIM
Git Clone & Build Radare2 Software
```



Raspberry Pi 4

https://www.adafruit.com/product/4292

64GB MicroSD Card

https://www.sparkfun.com/products/16498

Micro SD Card Reader/Writer

https://www.walmart.com/ip/Iogear-GFR204SD-SD-MicroSD-MMC-Card-Reader-and-Writer/15522266

Download 64-bit Kali Linux ARM Image

Kali Linux RaspberryPi 2 (v1.2), 3 and 4 (64-Bit) (img.xz)

https://www.offensive-security.com/kali-linux-arm-images

Download balenaEtcher

https://www.balena.io/etcher

Flash Kali ARM Image

OPTIONAL: Video [Load Kali RPI 4]

https://youtu.be/Jquf9BDm4iU

How To Install VIM

https://www.simplified.guide/ubuntu/install-vim

After obtaining all the necessary devices and software please watch the video on how to set up your environment as Null Byte did an amazing job with a step-by-step tutorial which will get you set-up in minutes.

The next step is to git clone and build the Radare2 software as this will we want the latest version as the standard version built into Kali will not be sufficient for our needs.

Git Clone & Build Radare2 Software

https://github.com/radareorg/radare2

```
cd Documents
git clone https://github.com/radareorg/radare2.git
sys/install.sh
```

Finally we will be using a text editor to build our code. Kali has both the VIM and Nano text editors built-in. We will be using VIM but you are free to use whatever one you are comfortable with.

In our next lesson we will write our first C++ program which will be "Hello World!".

Part 3 - "Hello World"

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to start at the beginning and take a very simple C++ program that does nothing more than use the stream insertion operator to send a string literal to the stdout and then use the end line manipulator to flush the output buffer.

Let's start by creating a file 0x01_asm64_helloworld.cpp and type the following into it.

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Let's compile and link.

```
g++ -o 0x01_asm64_helloworld 0x01_asm64_helloworld.cpp
```

Let's run.

```
./0x01_asm64_helloworld
```

We see the simple result.

```
Hello World!
```

These lessons are deliberately intended to be SHORT an SIMPLE. I know a number of you are more advanced however I really want to make this course as beginner friendly as possible.

In our next lesson we will debug this very simple binary using our dev build of Radare2.

Part 4 - Debugging "Hello World"

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to debug our first program utilizing our dev build of Radare2.

To begin let's open up our binary in Radare2.

```
radare2 ./0x01_asm_64_helloworld
```

Let's take advantage of Radare2's auto analysis feature.

```
aaa
```

The next thing we want to do logically is fire up the program in debug mode so it maps the raw machine code from disk to a running process.

```
ood
```

Now that we have a running instance we can seek to the main entry point of the binary.

```
s main
```

Let us take an initial examination by doing the following.

```
v
```

The output from Radare2 is entirely too large to display in this course however as you follow along in your own environment you will be able to follow along. We will keep this convention throughout this course for better readability of the document.

Remember there is a difference between an executable on disk and what resides when it is mapped. When it is on disk it is referred to as unmapped. We will look at that at the end of the lesson. For now we are looking at a mapped version as you see it is an offset of the mapped code we will examine later.

Do you notice that as your mapped memory values are different than mine? That is because of ALSR.

Address Space Layout Randomization (ASLR) is a security technique used in operating systems, first implemented in 2001. The current versions of all major operating systems (iOS, Android, Windows, macOS, and Linux) feature ASLR protection.

ASLR is primarily used to protect against buffer overflow attacks. In a buffer overflow, attackers feed a function as much junk data as it can handle, followed by a malicious payload.

We notice in my mapped memory that at address *0x55629cab48* we see our string *"Hello World!"*. You will have a different offset as we discussed but will find the same result.

Let us get back to a console window by doing the following.

```
q
```

Let's verify our initial analysis.

```
[0x55629ca9e4]> ps @0x55629cab48
Hello World!
[0x55629ca9e4]>
```

We can see that it is in fact true that at the mapped memory address of *0x55629cab48* we see the string *"Hello World!"*.

Let's also look at the hex view so we can see and better understand what is going on at the machine code level.

```
px @0x55629cab48
```



We see our "Hello World!" string and we can again see that it exists starting at the mapped memory address of 0x55629cab48.

We see that our machine code instructions are 16 bytes long or 64-bits long as we can see the first column start at *48* and end with *00*.

It is VERY important we understand a few key things. First is the fact that a single hex digit is 4-bits wide or a nibble or a half of a byte. In our case *4* is a half of a byte and *8* is the other half of the byte. Together they form a byte and in our case a valid ascii char code.

Let's visit the online ascii table.

http://www.asciitable.com

Second, we need to understand what the machine code translates to. Let's look up what *48* is in hex. We see that it is a capital 'H'. That maps perfectly as you see in the right hand column of the image above we see a 0 and below it the letter H.

Obviously *65* hex is ***'e'*** and so on and so forth. You can extrapolate the rest for yourself now that you have a basic understanding of what you are looking at.

Let's now define a breakpoint on main and execute this binary to verify in fact that when we continue on from main it will print *"Hello World"* to the stdout.

```
[0x55629ca9e4]> db 0x55629ca9e4
[0x55629ca9e4]>
```

Let us continue and verify our hypothesis. First we continue and break on main.

```
[0x55629ca9e4]> dc
hit breakpoint at: 0x55629ca9e4
[0x55629ca9e4]>
```

Now we step again and since there are no other breakpoints we will conclude the execution and verify our result in stdout.

```
[0x55629ca9e4]> dc
Hello World!
(59575) Process exited with status=0x0
[0x7fb146cb8c]>
```

Let's exit Radare2.

```
q
y
y
```

Let us rerun Radare2 again and this time not run the binary and simply look at the unmapped binary that is on disk.

```
radare2 ./0x01_asm_64_helloworld
```

Let's auto analyze.

```
aaa
```

Let's seek to main.

```
s main
```

Then view.

```
v
```

Notice that we have *"Hello World!"* this time at the unmapped memory address of *0xb48*. You notice that when you ran the binary the executable had an offset to this value but the LSB were *48* hex.

I hope this lesson helps you to understand the basics of 64-bit ARM assembly and how to reverse it properly.

In our next lesson we will hack the value.

Part 5 - Hacking "Hello World"

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

In the last lesson we spent a good deal of time really understanding what is going on inside our binary. This laid the groundwork for an easy hack.

Let's fire up radare2 in write mode.

```
radare2 -w ./0x01_asm_64_helloworld
```

Let's auto analyze.

```
aaa
```

Seek to main.

```
s main
```

View disassembly.

```
v
```

We see the memory addresses as they are on disk as we are not running the binary as we discussed in the last lesson.

We see that at **0xb48** we very easily find our string.

Let's get back to the terminal view.

```
q
```

Let's verify the string.

```
[0x000009e4]> ps @0xb48
Hello World!
[0x000009e4]>
```

Let's hack the string.

```
[0x000009e4]> w Hacked World @0xb48
```

Let's verify the hack.

```
[0x000009e4]> ps @0xb48
Hacked World
[0x000009e4]>
```

Let's quit radare2.

```
q
```

Now let's run our binary again!

```
./0x01_asm_64_helloworld
Hacked World
```

We see that we very easily hacked the binary. These lessons will help you understand how an attacker creates a workflow so you can learn how to anticipate and better reverse engineer.

In our next lesson we will work with simple I/O.

Part 6 - Basic I/O

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to look at a basic I/O C++ program that has some minimal validation.

Before I get into the brief lecture as I try to keep these short, I wanted to explain why I am not using the textbook straight *cin* examples that you see across the globe.

The *cin*, standard input stream, which takes input from the keyboard is referred to as our stdin.

What *cin* does is use whitespace, tab and newline as a terminator to the input stream. For example if you input *'abc'* and hit a tab or put a whitespace or newline by hitting return the data to the right of it will be ignored.

The problem is if you read from *cin* again it will pick up the remaining data in the stream if you do not flush the input buffer.

If you had for example:

```
std::cin >> val1;
std::cin >> val2;
```

If the user enters *1* and then leaves a space and then *2* and presses enter, you have no issue. *1* will be assigned into *val1* and *2* will be assigned to *val2* as they are chained.

The problem is what if you enter *'Hey Jude'* instead of an integer? What happens is it tries to read an integer and it goes into a failed state and from that point everything else it is extracting is unreliable.

I did not mean to be long winded but I really wanted to emphasize why you would NEVER use *cin* by itself and I mean NEVER!

Let's take a look at our basic i/o program that we will debug in the next lesson with a very basic C++ program that validates input.

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    int age = 0;
    bool valid = false;
    char null = '\0';

    while (!valid)
    {
        std::cout << "Enter Age: ";

        // Get input as string
        std::string line;
        getline(std::cin, line);

        // Init stringstream
        std::stringstream is(line);

        // Attempt to read a valid age from the
stringstream and
        // if a number can't be read, or there is more
than white
        // space in the string after the number, then
fail the read
        // and get another string from the user and make
sure the
        // dude is at least a year old and less than or
equal to
        // 100 years old
        if (!(is >> age) || (is >> std::ws &&
is.get(null)) || age >= 100 || age <= 0)
            std::cout << "Dude be real!" << std::endl;
        else
            valid = true ;
    }

    std::cout << "Your are " << age << " years old, seems
legit!" << std::endl;

    return 0;
}
```

We start by importing *iostream*, *sstream* and *string*. So far nothing tricky.

We then prompt the user to enter their age. We then create a string object called *line* and take advantage of C++ *getline()* which is a standard C++ library function that is used to read a string or a line from an input stream properly.

We then take advantage of the *stringstream* as it associates a string object with a stream allowing you to read from the string as if it were a stream like we would do with raw *cin*. In this simple example we create an *is* object which is short for input stringstream and connect it with our *line* object.

Then before we echo data to stdout we do a little validation. We first check to see if *age* is the type it was defined as which is an *int* OR is there a white space in the stream after a given integer OR is age greater than *100* or less than *0*. Very simply it provides a response if the input does not meet this criteria.

Finally if all is well it echoes out a simple *cout*.

Let's compile and link.

```
g++ -o 0x02_asm64_basicio 0x02_asm64_basicio.cpp
```

Let's run.

```
./0x02_asm64_basicio
```

Depending on what you enter it will validate as appropriate as described above. PLEASE try this example and manipulate the source to get a full understanding of what is going on here.

In our next lesson we will debug this very simple binary using our dev build of Radare2.

Part 7 - Debugging Basic I/O

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to debug our very basic input validation program from last lecture.

To begin let's open up our binary in Radare2.

```
radare2 ./0x02_asm_64_basicio
```

Let's take advantage of Radare2's auto analysis feature.

```
aaa
```

The next thing we want to do logically is fire up the program in debug mode so it maps the raw machine code from disk to a running process.

```
ood
```

Now that we have a running instance we can seek to the main entry point of the binary.

```
s main
```

Let us take an initial examination by doing the following.

```
v
```

A couple things to note we see at *0x5566be00cc* the output of *"Enter Age: "* and at *0x5566be017c* a call to *istream* which is going to capture the values from *stdin* to which we identify a failure condition at 0x5566be01d0 where we find *"Dude be real!"* and we see the results of a proper input validation starting at *0x5566be0218* _*where we say* "You are " *and then we see a call to the output stream at* _*0x5566be0238* and then the continuation of the validation string at *0x5566be0244* where we say *" years old, seems legit!"*.

The next step is to look at the binary with a visual graph.

```
q
VV
ppppp
```

This is our zoomed out visual graph. We can see how the program moves from function to function. You will notice there are a series of tags such as [ol] or [ok] and you can literally type the following:

```
p
ol
```

Now we are inside that function.

Then to go back to main.

```
qq
s main
VV
```

This will take us to an expanded graph that we can also use our arrow keys to look around.

Let's set a breakpoint at *0x5566be00c4* where we *bne 0x5566be0214* which is where we see the success route of our binary.

```
[0x5566be0194]> db 0x5566be00c4
[0x5566be0194]> dc
hit breakpoint at: 0x5566be00c4
Enter Age: 33
hit breakpoint at: 0x5566be00c4
[0x5566be0194]> dc
Your are 33 years old, seems legit!
(2215) Process exited with status=0x0
```

As you can see we cycled the loop and entered in a correct validation and was able to get our success return.

In our next lesson we will hack the validation.

Part 8 - Hacking Basic I/O

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we hack the input validation from our last lesson.

Let's fire up radare2 in write mode.

```
radare2 -w ./0x02_asm_64_basicio
```

Let's auto analyze.

```
aaa
```

Seek to main.

```
s main
```

View disassembly.

```
v
```

Let's get back to the terminal view.

```
q
```

Let's look at the visual graph and begin with the first *b.ne* which under the proper expected conditions it will only accept a valid integer between *0* and *100* as we demonstrated in the last lecture.

The *b.ne* meaning *branch if not equal*. The assembly before it simply does not matter in this case as we know if we leave b.ne as is the input validation will be in tact.

We need to disable this input validation by changing that instruction to a *b.eq* or *branch if equal*.

Let's look at that code block.



We see that it if it is true, meaning validation is correct and we have an integer between 0 and 100 we will follow the true green line to the next function.

If we fail the validation we will be sent to the false condition to obtain new input.

Let's q to a terminal prompt.

```
qq
```

Let's seek to the statement we want to hack.

```
[0x000010a4]> s 0x000010c4
```

Let's now hack the branch as discussed.

```
[0x000010c4]> wa b.eq 0x1214
Written 4 byte(s) (b.eq 0x1214) = wx 800a0054
[0x000010c4]>
```

Let's quit.

```
q
```

Now when we run the binary it will simply ignore any input at all let alone input validation and simply arrive at the desired point.

```
kali@kali:~/Documents/0x02_asm_64_basicio$
./0x02_asm_64_basicio
Your are 0 years old, seems legit!
kali@kali:~/Documents/0x02_asm_64_basicio$
```

Even though 0 is valid it is simply an unstable value that happened to be in one of the registers that the program expected to be properly assigned during a normal program flow. Here we were able to change the binary permanently to accomplish our hack.

These are VERY simple examples however when you combine these as you progress you will literally be able to Reverse Engineer anything.

In our next lesson we will discuss the char primitive data type.

Part 9 - Character Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to talk about the first of the C++ primitive. The *char* datatype is used to store a single character and must be surrounded by single quotes.

Let's look at our basic example.

```
#include <iostream>

int main()
{
    char my_char = 'c';

    std::cout << my_char << std::endl;

    return 0;
}
```

Extremely simple. We are simply creating a char variable called *my_char _and assigning it the character _c*.

We then print it to stdout and nothing more.

Let's compile and link.

```
g++ -o 0x03_asm64_char_primitive_datatype
0x03_asm64_char_primitive_datatype.cpp
```

Let's run.

```
./0x03_asm64_char_primitive_datatype
```

Very simply we see the following.

```
c
```

It successfully echoed *c* to the terminal stdout. Very simple.

Next week we will debug this very simple example.

Part 10 - Debugging Character Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to debug our very simple character primitive datatype.

To begin let's open up our binary in Radare2.

```
radare2 ./0x03_asm64_char_primitive_datatype
```

Let's take advantage of Radare2's auto analysis feature.

```
aaa
```

The next thing we want to do logically is fire up the program in debug mode so it maps the raw machine code from disk to a running process.

```
ood
```

Now that we have a running instance we can seek to the main entry point of the binary.

```
s main
```

Let us take an initial examination by doing the following.

```
v
```

We can see that at *0x5576bff9ec* we are moving *0x63* or ascii *'c'* into the *w0* register. REMEMBER your address will be different due to ASLR.

Let's set a breakpoint at *0x5576bff9ec* and verify the contents.

```
[0x5576bff9e4]> db 0x5576bff9ec
[0x5576bff9e4]> dc
hit breakpoint at: 0x5576bff9ec
[0x5576bff9ec]> dr w0
0x00000001
[0x5576bff9ec]> ds
[0x5576bff9ec]> dr w0
0x00000063
[0x5576bff9ec]>
```

This is very simple but let's break it down. We set our breakpoint and continued. We looked inside the register w0 and saw that the value is 0x01.

We then stepped once and looked again to see that 0x63 was successfully moved into w0 as now we see it does in fact contain 0x63.

If we dc again we see it echoed to the stdout as expected.

```
[0x5576bff9ec]> dc
c
(10845) Process exited with status=0x0
[0x7f9727503c]>
```

In our next lesson we will hack the char to another value of our choice.

Part 11 - Hacking Character Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we hack the char from the last lesson.

Let's fire up radare2 in write mode.

```
radare2 -w ./0x03_asm64_char_primitive_datatype
```

Let's auto analyze.

```
aaa
```

Seek to main.

```
s main
```

View disassembly.

```
v
```

Let's get back to the terminal view.

```
q
```

All we have to do is write assembly to *0x000009ec* and specify a new char of our choosing.

```
[0x000009e4]> wa movz w0, 0x66 @ 0x000009ec
Written 4 byte(s) (movz w0, 0x66) = wx c00c8052
[0x000009e4]>
```

Let's quit and run the new binary from the terminal.

```
[0x000009e4]> q
kali@kali:~/Documents/0x03_asm64_char_primitive_datatype$
./0x03_asm64_char_primitive_datatype
f
```

As you can see we successfully and permanently hacked the binary! It is very trivial but when you take the last series of lessons together with each new successive lesson you build a real skill-set!

In our next lesson we will work with the boolean primitive datatype.

Part 12 - Boolean Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to talk about the C++ *boolean* datatype that stores either a *0* or *1* to represent *0* for *false _and _1* for anything *true*.

This kind of flag is used extensively in programming in general and we will look at another very basic program to understand its simple usage.

```
#include <iostream>

    int main()
    {
        bool my_bool = true;

        std::cout << my_bool << std::endl;

        return 0;
    }
```

We see that we are creating a *bool* and assigning it a _true _value or _1 _value and printing it.

Let's compile and link.

```
g++ -o 0x04_asm64_boolean_primitive_datatype
0x04_asm64_boolean_primitive_datatype.cpp
```

Let's run.

```
./0x04_asm64_boolean_primitive_datatype
```

We simply see the following.

```
1
```

It successfully echoed *1* to the terminal stdout. Very simple.

Next week we will debug this very simple example.

Part 13 - Debugging Boolean Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to debug our very simple boolean primitive datatype.

To begin let's open up our binary in Radare2.

```
radare2 ./0x04_asm64_boolean_primitive_datatype
```

Let's take advantage of Radare2's auto analysis feature.

```
aaa
```

The next thing we want to do logically is fire up the program in debug mode so it maps the raw machine code from disk to a running process.

```
ood
```

Now that we have a running instance we can seek to the main entry point of the binary.

```
s main
```

Let us take an initial examination by doing the following.

```
v
```

We see in *0x55718999bc movz w0, 0x1_or moving _0x1* into *w0* which is our bool true. REMEMBER your address will be different due to ASLR.

Let's set a breakpoint at *0x55718999bc* and verify the contents.

```
[0x55718999b4]> db 0x55718999bc
[0x55718999b4]> dc
```

```
hit breakpoint at: 0x55718999bc
```

```
[0x55718999bc]> ds
[0x55718999bc]> dr w0
0x00000001
[0x55718999bc]>
```

Very simply we broke right before the value *0x1* was to be placed in *w0* and then we stepped and saw that it was in fact *0x1* inside of *w0* after the step. This means that our program successfully put a *1 _or _true* into the *w0* register which matches what our source code created.

If we dc again we see it echoed to the stdout as expected.

```
[0x55718999bc]> dc
1
(96445) Process exited with status=0x0
```

```
[0x7fac4f903c]>
```

In our next lesson we will hack the boolean to make it 0.

Part 14 - Hacking Boolean Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we hack the boolean from the last lesson.

Let's fire up radare2 in write mode.

```
radare2 -w ./0x04_asm64_boolean_primitive_datatype
```

Let's auto analyze.

```
aaa
```

Seek to main.

```
s main
```

View disassembly.

```
v
```

Let's get back to the terminal view.

```
q
```

All we have to do is write assembly to *0x00000009bc* and specify *0x0*.

```
[0x000009b4]> wa movz w0, 0x0 @ 0x00000009bc
Written 4 byte(s) (movz w0, 0x0) = wx 00008052
```

```
[0x000009b4]>
```

Let's quit and run the new binary from the terminal.

```
[0x000009b4]> q
kali@kali:~/Documents/0x04_asm64_boolean_primitive_dataty
pe$ ./0x04_asm64_boolean_primitive_datatype
```

```
0
```

As you can see we successfully and permanently hacked the binary! What was originally *true* or *1* is now *false _or _0*.

In our next lesson we will work with the integer primitive datatype.

Part 15 - Float Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to talk about the C++ *float* datatype that stores floating point values.

```
#include <iostream>

int main()
{
    float my_float = 10.1;

    std::cout << my_float << std::endl;

    return 0;
}
```

Very simply we create a float and assign a simple value to it and print it.

Let's compile and link.

```
g++ -o 0x05_float_primitive_datatype
0x05_float_primitive_datatype.cpp
```

Let's run.

```
./0x05_float_primitive_datatype
```

We simply see the following.

```
10.1
```

It successfully echoed *10.1* to the terminal stdout. Very simple.

Next week we will debug this very simple example.

Part 16 - Debugging Float Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to debug our very simple float primitive datatype.

To begin let's open up our binary in Radare2.

```
radare2 ./0x05_asm64_float_primitive_datatype
```

Let's take advantage of Radare2's auto analysis feature.

```
aaa
```

The next thing we want to do logically is fire up the program in debug mode so it maps the raw machine code from disk to a running process.

```
ood
```

Now that we have a running instance we can seek to the main entry point of the binary.

```
s main
```

Let us take an initial examination by doing the following.

```
v
```

When dealing with floating point numbers in ARM64 we have to understand that we want to locate where the *fmov* instruction occurs where we take a value from our *w0* register and move it into the floating point *s0* register. Here is where all the magic happens!

Let us define a break point right below the *fmov* instruction. REMEMBER with ASLR your addresses will be different than this example.

```
[0x557931c9b4]> db 0x557931c9c8
[0x557931c9b4]> dc
[0x557931c9b4]> hit breakpoint at: 0x557931c9c8
[0x557931c9c8]> ds
[0x557931c9c8]> dr w0
0x4121999a
[0x557931c9c8]>
```

OK so we see this strange value which if you look at the code below, the *lsl* which is logical shift left, is moving the byte order of which we are using the *movz* and *movk* instructions which *movz* will move *0x999a* into *w0* and then the *movk* will move *0x4121, lsl 16* into *w0* therefore putting 4121 at the higher order byte locations and the 999a at the lower order byte locations.

```
movz w0, 0x999a
movk w0, 0x4121, lsl 16
fmov s0, w0
```

We move our *w0* register into *s0* so we HAVE to change these values here before letting it get into *s0* otherwise it will be significantly harder to hack in the next lesson.

Lets continue to show our value.

```
[0x557931c9c8]> dc
10.1
(237691) Process exited with status=0x0
[0x7fb948407c]>
```

In our next lesson we will hack this value!

Part 17 - Hacking Float Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we hack the float from the last lesson.

First update our radare2 source code.

```
cd radare2
git pull
sys/user.sh
```

If you did not follow the instructions earlier you have to build radare2 from source for this to work as they rarely update releases.

https://github.com/radareorg/radare2

If you do not have the repo, clone it and follow the instructions above.

Let's fire up radare2 in write mode.

```
radare2 -w ./0x05_asm64_float_primitive_datatype
```

Let's auto analyze.

```
aaa
```

Seek to main.

```
s main
```

View disassembly.

```
v
```

Let's get back to the terminal view.

```
q
```

We need to hack two instructions here. Let's examine two very specific instructions.

```
movz w0, 0x999a
movk w0, 0x4121, lsl 16
```

Remember from last week that ultimately w0 is going to hold *0x4121999a* as the *lsl* moves the bites in reverse byte order.

Currently this will produce a float of *10.1* as we have seen in the prior lessons. It is critical that you understand that in floating-point numbers there is a *mantissa* which in our case is *10* and an *exponent* which is the *1* to which they are separated by a *.* which ties them together.

Therefore to get *10.2* we would need to write assembly and update these instructions.

```
[0x000009b4]> wa movz w0, 0x3333 @0x000009bc
[0x000009b4]> wa movk w0, 0x4123, lsl 16 @0x000009c0
q
```

Now run the binary!

```
kali@kali:~/Documents/0x05_float_primitive_datatype$
./0x05_float_primitive_datatype
10.2
```

I want you to take a close look at some examples I have put together for you so that you can understand how different values result in different results. Keep in mind these results are in an active debug session so the addresses will be different so your ASLR will have different values.

```
[0x555e6c29c4]> dr w0 = 0x4122999a
0x4121999a ->0x4122999a
[0x555e6c29c4]> dc
hit breakpoint at: 0x555e6c29c8
[0x555e6c29c8]> dc
10.1625
(238252) Process exited with status=0x0

[0x556215e9c4]> dr w0 = 0x41235555
0x4121999a ->0x41235555
[0x556215e9c4]> dc
hit breakpoint at: 0x556215e9c8
[0x556215e9c8]> dc
10.2083
(238258) Process exited with status=0x0

[0x558216c9c4]> dr w0 = 0x4123599a
0x4121999a ->0x4123599a
[0x558216c9c4]> dc
hit breakpoint at: 0x558216c9c8
[0x558216c9c8]> dc
10.2094
(238257) Process exited with status=0x0

[0x55868a79c4]> dr w0 = 0x4123999a
0x4121999a ->0x4123999a
[0x55868a79c4]> dc
hit breakpoint at: 0x55868a79c8
[0x55868a79c8]> dc
10.225
(238253) Process exited with status=0x0

[0x55826479c4]> dr w0 = 0x41233333
0x4121999a ->0x41233333
[0x55826479c4]> dc
hit breakpoint at: 0x55826479c8
[0x55826479c8]> dc
10.2
(238259) Process exited with status=0x0

[0x55716ab9c4]> dr w0 = 0x4125999a
0x4121999a ->0x4125999a
[0x55716ab9c4]> dc
hit breakpoint at: 0x55716ab9c8
[0x55716ab9c8]> dc
10.35
(238250) Process exited with status=0x0
```

```
[0x55880169c4]> dr w0 = 0x412f999f
0x4121999a ->0x412f999f
[0x55880169c4]> dc
hit breakpoint at: 0x55880169c8
[0x55880169c8]> dc
10.975
(238245) Process exited with status=0x0

[0x559130d9c4]> dr w0 = 0x412ff99e
0x4121999a ->0x412ff99e
[0x559130d9c4]> dc
hit breakpoint at: 0x559130d9c8
[0x559130d9c8]> dc
10.9984
(238246) Process exited with status=0x0

[0x557b1b39c4]> dr w0 = 0x412fff9e
0x4121999a ->0x412fff9e
[0x557b1b39c4]> dc
hit breakpoint at: 0x557b1b39c8
[0x557b1b39c8]> dc
10.9999
(238247) Process exited with status=0x0

[0x55931439c4]> dr w0 = 0x412ffffe
0x4121999a ->0x412ffffe
[0x55931439c4]> dc
hit breakpoint at: 0x55931439c8
[0x55931439c8]> dc
11
(238248) Process exited with status=0x0
```

You can start to see patterns here. TAKE THE TIME AND ACTUALLY TRY THESE OUT so you have a better understand of how these values ultimately go into the s0 register!

Next lesson we will discuss doubles.

Part 18 - Double Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to talk about the C++ *double* datatype that stores double floating point values.

```
#include <iostream>

int main()
{
    double my_double = 10.1;

    std::cout << my_double << std::endl;

    return 0;
}
```

Very simply we create a float and assign a simple value to it and print it.

Let's compile and link.

```
g++ -o 0x06_double_primitive_datatype
0x05_double_primitive_datatype.cpp
```

Let's run.

```
./0x06_double_primitive_datatype
```

We simply see the following.

```
10.1
```

It successfully echoed *10.1* to the terminal stdout. Very simple.

Next week we will debug this very simple example.

Part 19 - Debugging Double Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we are going to debug our very simple double primitive datatype.

To begin let's open up our binary in Radare2.

```
radare2 ./0x06_asm64_double_primitive_datatype
```

Let's take advantage of Radare2's auto analysis feature.

```
aaa
```

The next thing we want to do logically is fire up the program in debug mode so it maps the raw machine code from disk to a running process.

```
ood
```

Now that we have a running instance we can seek to the main entry point of the binary.

```
s main
```

Let us take an initial examination by doing the following.

```
v
```

When dealing with double floating-point numbers in ARM64 we have to understand that we want to locate where the *fmov* instruction occurs where we take a value from our *w0* register and move it into the floating point *d0* register. Here is where all the magic happens! This is just like our floating-point numbers that deal with *s0*.

Let us define a break point right below the *fmov* instruction. REMEMBER with ASLR your addresses will be different than this example.

```
[0x556bf809b4]> db 0x556bf809c4
[0x556bf809b4]> dc
hit breakpoint at: 0x556bf809c4
[0x556bf809c4]> dr w0
0x33333333
```

We move our *w0* register into *d0* so we HAVE to change these values in *d0* which is different from our float. We will explore this in the next lesson.

Lets continue to show our value.

```
[0x556bf809c4]> dc
10.1
(39979) Process exited with status=0x0
[0x7fa37da0fc]>
```

In our next lesson we will hack this value!

Part 20 - Hacking Double Primitive Datatype

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.
https://github.com/mytechnotalent/hacking_c-_arm64

Today we hack the double from the last lesson.

Let's fire up radare2 in write mode.

```
radare2 -w ./0x06_asm64_double_primitive_datatype
```

Let's auto analyze.

```
aaa
```

Seek to main.

```
s main
```

View disassembly.

```
v
```

Let's get back to the terminal view.

```
q
```

All we have to do now is write the new value of d0 into the register where the fmov instruction is and quit.

```
wa mov x0, 0x6666666666666666 @0x000009bc
q
```

Then we run our new binary.

```
kali@kali:~/Documents/0x06_double_primitive_datatype$
./0x06_asm64_double_primitive_datatype
```

```
10.2
```

I hope you enjoyed this series and have a good firm grasp on ARM64 RE!

Pico Hacking Course

Let's dive in rightaway!

Part 1 - The Why, The How...

It is 2021 and here we are once again covering a new Reverse Engineer course. This course will focus on the C programming language to which we will statically reverse the compiled ARM 32 elf binary utilizing the Radare2 debugger on a Raspberry Pi Pico microcontroller.

What are microcontrollers? We can find them in vehicles, robots, office machines, medical devices, mobile radio transceivers, vending machines and home appliances, among other devices. They are targeted machines designed to control small features of a larger component, without a complex front-end operating system.

We will be writing very basic C programs and then reverse them one at a time in ARM 32 Assembly.

I am going to assume you are working with an Ubuntu Linux distro...

You will first need a Raspberry Pi Pico.

You will need the Radare2 repo.

```
git clone https://github.com/radareorg/radare2.git
cd radare2
cd radare2 sys/install.sh
```

You NEED to build from source! The versions that are packaged in Ubuntu and Kali Linux are older and do not have the features we require for our level of reversing.

You will need VIM.

```
sudo apt install vim
```

You will need to update .vimrc file.

```
vim ~/.vimrc
```

Then...

```
set number
set tabstop=2
set shiftwidth=2
set expandtab
syntax on
set syntax=c
```

You will need the Raspberry Pi Pico repo.

```
mkdir pico
cd pico
git clone -b master https://github.com/raspberrypi/pico-
sdk.git
cd pico-sdk
git submodule update --init
cd ..
git clone -b master https://github.com/raspberrypi/pico-
examples.git
sudo apt update
sudo apt install cmake gcc-arm-none-eabi libnewlib-arm-
none-eabi build-essential
```

Let's build the blink program.

```
cd pico-examples
mkdir build
cd build
export PICO_SDK_PATH=../../pico-sdk
cmake ..
cd blink
make
```

Copy the **blink.uf2** file to your Pico.

Congrats you got a blinking C program!

In our next lesson we will create a simple, "Hello, World" program.

Part 2 - Hello World

Today we are going to cover the basic setup for creating our own projects on the Raspberry Pi Pico.

Inside of our **pico** folder lets create a **0x02_pico_hello_world** folder alongside of the **pico-sdk** and **pico-example** folders.

```
mkdir 0x02_pico_hello_world
cd 0x02_pico_hello_world
```

Let's create our vim **0x02_hello_world.c** file.

```
vim 0x02_hello_world.c
```

Let's add the following.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    printf("Hello world!\n");

    sleep_ms(1000);
  }

  return 0;
}
```

We first handle the logic to init all standard input and output.

```
    stdio_init_all();
```

Finally we print *"Hello world!"* every 1 second to the standard output in an infinite loop.

```
    while(1)
    {
      printf("Hello world!\n");

      sleep_ms(1000);
    }
```

We then upon success *return 0* to indicate success as our *main* function is an int. It is not technically required but good practice.

```
    return 0;
```

Working with **cmake** significantly helps in the build process for our projects. We first need to make a **CMakeLists.txt** file.

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(0x02_hello_world
  0x02_hello_world.c
)

pico_enable_stdio_usb(0x02_hello_world 1)

pico_add_extra_outputs(0x02_hello_world)

target_link_libraries(0x02_hello_world pico_stdlib)
```

Next we need to copy the **pico_sdk_import.cmake** file from the external folder in the **pico-sdk** installation to the **0x02_hello_world** project folder.

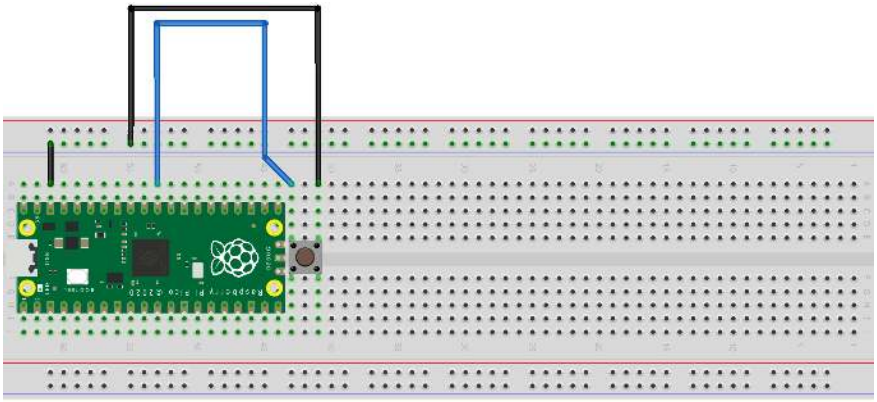
```
cp ../pico-sdk/external/pico_sdk_import.cmake .
```

Finally we are ready to build.

```
mkdir build
cd build
export PICO_SDK_PATH=../../pico-sdk
cmake ..
make
```

This will produce a number of files and the ones we are going to focus on are the **.elf** file when it comes to debugging and hacking which is the full program output, possibly including debug information and the **.uf2** file which is the program code and data in a UF2 form that you can drag-and-drop on to the RP2040 board when it is mounted as a USB drive.

I took the time to wire up a reset button on the Pico so that I do not have to keep unplugging in the USB and pressing the BOOTSEL every time I need to re-flash so here is the schematic of such.



To flash press the external button and while it is still pressed, press the BOOTSEL on the board, then release the BOOTSEL and finally release the external button.

Then simply copy the **.uf2** file to the drive.

```
cp 0x02_hello_world.uf2 /Volumes/RPI-RP2
```

Then we need to locate the USB drive so you can do the following.

```
ls /dev/tty.
```

Press tab to find the drive and then in my case I will use **screen** to connect.

```
screen /dev/tty.usbmodem0000000000001
```

Hooray! You should see, "Hello world!" to the standard output every second.

In our next lesson we will debug the **.elf** binary in **Radare2**.

Part 3 - Debugging Hello World

Today we will dive into debugging our very simple, "Hello world!", program.

Let's review our code.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    printf("Hello world!\n");

    sleep_ms(1000);
  }

  return 0;
}
```

Please make sure you build Radare2 from source. Before each lesson PLEASE complete the following.

```
git pull
radare2 sys/install.sh
```

You can check that the version is up to date.

```
radare2 -v
```

In my case, as it will be different for you.

```
radare2 5.2.0-git 25988 @ darwin-x86-64 git.5.1.1
commit: 510ddab0e523bed173b3954e5f61abf395812f7d build:
2021-03-21__05:40:51
```

Now back to our project repo. Let's fire up our debugger.

```
radare2 -w arm -b 16 0x02_hello_world.elf
```

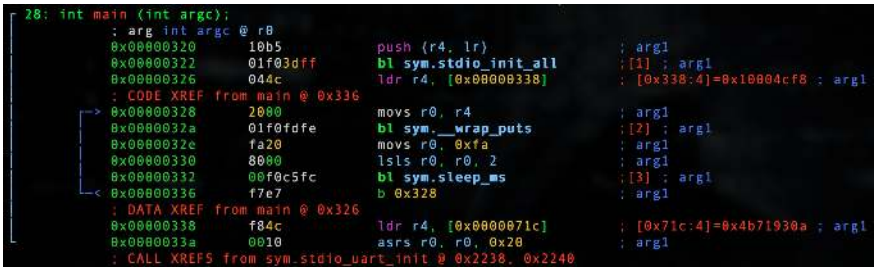
Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



Let's break this very simple program down.

```
push {r4, lr}
```

We are simply setting up our function arguments where we pushing the value of *r4* and *lr* (link register) to the stack.

We then bl (branch long) to the *sym.stdio_init_all* function which init's standard input and output.

```
bl sym.stdio_init_all
```

We then load the value at the location *0x00000338* into the *r4* register. This is where the, **"Hello world!"** string lives.

```
ldr r4, [0x00000338]
```

To prove this we can do the following by pressing : inside of the current Visual mode and then typing the following.

```
:> psz @ [0x00000338]
Hello world!
:> psz @ 0x00004cf8
Hello world!
```

As you can clearly see the value inside of *0x00000338 _is the value at _0x0004cf8*.

We then move and set the flags (that is the *s* in *movs*) the contents of *r4* into *r0*.

```
movs r0, r4
```

We then branch long to the puts wrapper. The debugger converted our _printf _function in our code to this wrapper function.

```
bl sym.__wrap_puts
```

We then *movs _250 decimal, 0xfa hex, which is 1/4 our 1000 millisecond sleep into _r0*.

```
movs r0, 0xfa
```

We then logically shift left, 2, and set the flags. This of course multiplies our 250 value by 2 and then again by 2 which takes 250 decimal to 1000 decimal which is our millisecond delay and places that 1000 decimal value into *r0*.

```
lsls r0, r0, 2
```

If you are not familiar with ARM 32 Assembly instructions, please reference this great table provided by Keil.

https://developer.arm.com/documentation/ddi0210/c/Introduction/Instruction-set-summary/ARM-instruction-summary?lang=en

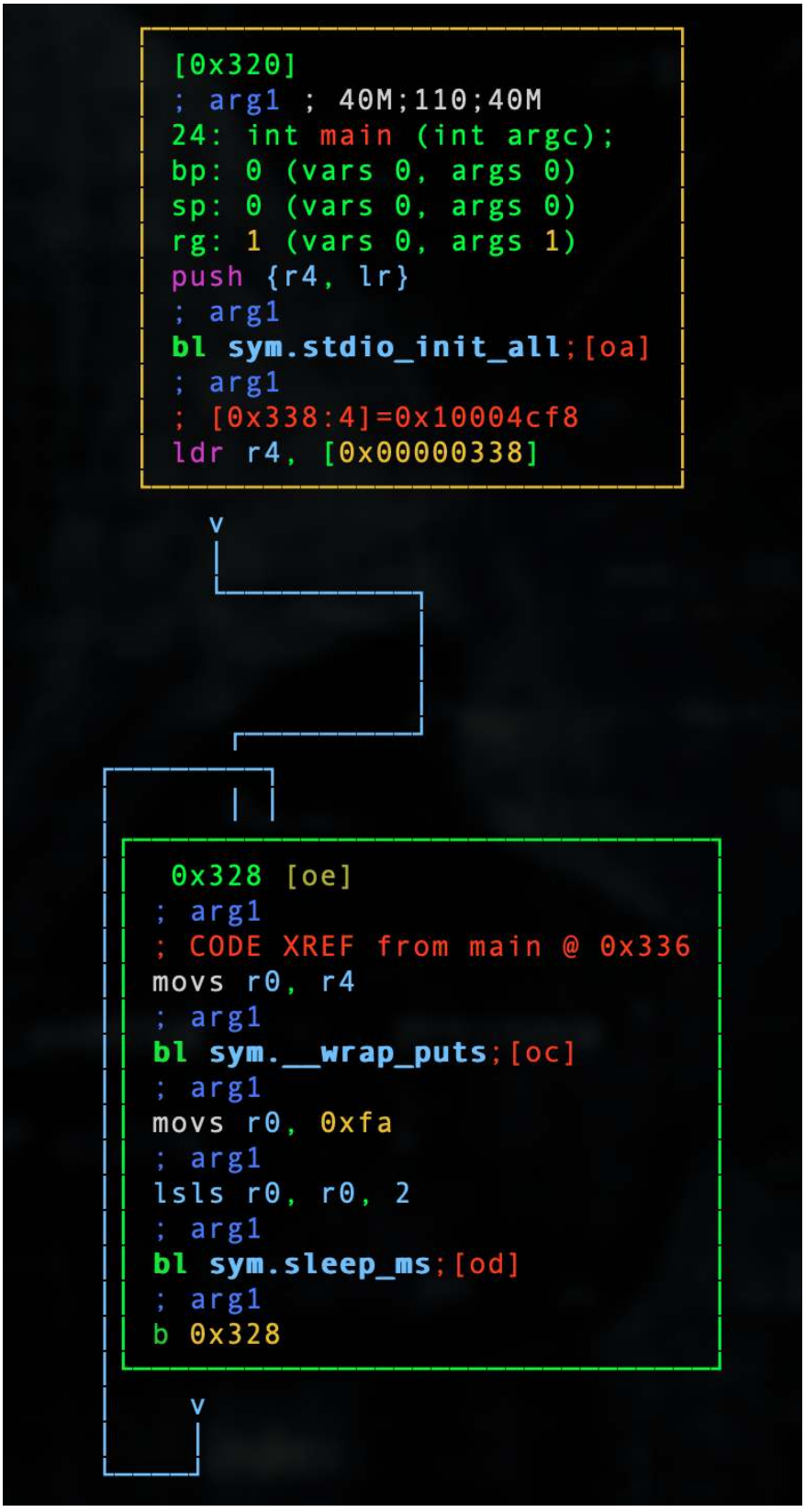
We then branch long to our *sleep_ms* function.

```
bl sym.sleep_ms
```

We then branch unconditional back to *0x328* which is our while loop.

```
b 0x328
```

You can also see the graph view by pressing **V** again in the current window.



This is a great way to trace through more elaborate code. I wanted to show you all this as you can use this going forward as you do larger analysis.

In our next lesson we will hack our simple program and convert it back to a **.uf2** and re-flash to the Pico.

Part 4 - Hacking Hello World

In the last lesson we reviewed how to properly debug our very simple binary in **Radare2**. Today we are going to hack that static **.elf** binary and convert it to the **.uf2** format and flash to our Pico and see the magic happen.

Let's review our very simple program once more.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    printf("Hello world!\n");

    sleep_ms(1000);
  }

  return 0;
}
```

Let's load up our binary.

```
radare2 -w arm -b 16 0x02_hello_world.elf
```

Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's use Visual mode and press p twice to get our our favorite debugger view.

```
V
```

Let's review the simple ARM32 Assembly.



I would hack this binary in two ways. As we discussed in the last lesson we see the contents inside the memory location *0x00000338* holding the value of our string. Let's press the colon : and press enter.

```
:> psz @ [0x00000338]
Hello world!
```

Let's review our strings. I want you to pay attention to the, "Hello world!" as you will see two addresses. The one on the left is the physical address and the one directly to the right is the virtual address. We will be concerned with the virtual address. To better understand let's do the following.

```
:> iz~ | less
```

As you can see our string is at the top.

```
[Strings]
nth paddr      vaddr      len size section type    string
————————————————————————————————————————————————————
0   0x00014cf8 0x00004cf8 12  13   .rodata ascii   Hello
world!
1   0x00014d08 0x00004d08 26  27   .rodata ascii   No
spinlocks are available
2   0x00014d24 0x00004d24 33  34   .rodata ascii
 Hardware alarm %d already claimed
3   0x00014d48 0x00004d48 15  16   .rodata ascii   \n***
PANIC ***\n
4   0x00014d5c 0x00004d5c 11  12   .rodata ascii   Hard
assert
5   0x00014d68 0x00004d68 7   8    .rodata ascii
 Release
6   0x00014d70 0x00004d70 5   6    .rodata ascii   1.0.0
7   0x00014d78 0x00004d78 4   5    .rodata ascii   pico
8   0x00014d80 0x00004d80 16  17   .rodata ascii
 0x02_hello_world
9   0x00014d94 0x00004d94 11  12   .rodata ascii   Mar 21
2021
10  0x00014db2 0x00004db2 4   5    .rodata ascii   uBhM
11  0x00014dbc 0x00004dbc 10  11   .rodata ascii   UART
stdin
12  0x00014dc8 0x00004dc8 11  12   .rodata ascii   UART
stdout
13  0x00014dd4 0x00004dd4 19  20   .rodata ascii   UART
stdin / stdout
14  0x00014dfc 0x00004dfc 18  19   .rodata ascii   USB
stdin / stdout
15  0x00014e1c 0x00004e1c 12  13   .rodata ascii
 Raspberry Pi
16  0x00014e2c 0x00004e2c 4   5    .rodata ascii   Pico
17  0x00014e34 0x00004e34 12  13   .rodata ascii
 000000000000
18  0x00014e44 0x00004e44 9   10   .rodata ascii   Board
CDC
19  0x00014ec4 0x00004ec4 19  20   .rodata ascii
 Unhandled IRQ 0x%x\n
20  0x00014ed8 0x00004ed8 39  40   .rodata ascii
 Isochronous wMaxPacketSize %d too large
21  0x00014f00 0x00004f00 30  31   .rodata ascii   ep %d
%s was already available
22  0x00014f20 0x00004f20 40  41   .rodata ascii   Can't
continue xfer on inactive ep %d %s
23  0x00014f4c 0x00004f4c 35  36   .rodata ascii
 Transferred more data than expected
0   0x00020135 0x10000135 5   6    .data   ascii
```

```
 V\n`\eh
1   0x0002018b 0x1000018b 5   6    .data   ascii   &CF\eh
2   0x000201a0 0x100001a0 4   5    .data   ascii   CF\ey
3   0x000201a8 0x100001a8 4   5    .data   ascii   CF\eh
4   0x000201d0 0x100001d0 4   5    .data   ascii   \thAq
5   0x0002028d 0x1000028d 5   6    .data   ascii   GpF\t8
6   0x00020805 0x10000805 5   11   .data   utf16le \a \b
\b
7   0x00020905 0x10000905 5   11   .data   utf16le \b \t
\t
8   0x00020a05 0x10000a05 5   11   .data   utf16le \t \n
\n
9   0x00020b05 0x10000b05 5   11   .data   utf16le \n \v
\v
(END)
```

You can see the value of *0x00004cf8* holds our string to prove it we can do the following.

```
:> psz @ 0x00004cf8
Hello world!
```

Let's hack this.

```
:> w Hacked World! @ [0x00000338]
```

Let's now verify the value is changed.

```
:> psz @ 0x00004cf8
Hacked World!
```

The other thing I would like to hack is the sleep_ms which is currently set at 1000. Remember it is showing 250 decimal or 0xfa hex and we logical shift left twice as we discuss in the last lesson. The first logical shift left will multiply by 2 bringing us to 500 and the 2nd logical shift left will multiply by 2 brining us to 1000.

```
lsls r0, r0, 2
```

Let's hack this by changing the 2 to a 1. This will make the delay 500 ms or a half a second.

```
:> wa lsls r0, r0, 1 @ 0x00000330
Written 2 byte(s) (lsls r0, r0, 1) = wx 4000
```

Let's verify.

```
:> pd 1 @ 0x00000330
│           0x00000330      4000           lsls r0, r0, 1
```

We can clearly see it changed.

All we have to do now is exit and convert our **.elf** to **.uf2**!

```
./elf2uf2/elf2uf2 0x02_hello_world.elf
0x02_hello_world.uf2
```

Plug in the Pico and make sure you hold down BOOTSEL or use the setup I provided in the last lesson.

```
cp 0x02_hello_world.uf2 /Volumes/RPI-RP2
```

Let's screen it!

```
screen /dev/tty.usbmodem0000000000001
```

AHH yea!

```
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
Hacked World!
```

Every half a second!

Next lesson we will discuss variables.

Part 5 - char

Today we will begin our coverage of the C data types. We will start with char. A char is the smallest addressable unit of the machine that can contain basic character set. It is an integer type and can be either can be either signed or unsigned.

Let's make a new dir **0x03_char** and add our **CMakeLists.txt** file in it.

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(0x03_char
  0x03_char.c
)

pico_enable_stdio_usb(0x03_char 1)

pico_add_extra_outputs(0x03_char)

target_link_libraries(0x03_char pico_stdlib)
```

Next we need to copy the **pico_sdk_import.cmake** file from the external folder in the **pico-sdk** installation to the **0x03_char** project folder.

```
cp ../pico-sdk/external/pico_sdk_import.cmake .
```

Let's create our C file **0x03_char.c** and roll...

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    char x = 'x';

    printf("%c\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Finally we are ready to build.

```
mkdir build
cd build
export PICO_SDK_PATH=../../pico-sdk
cmake ..
make
```

Then simply copy the **.uf2** file to the drive.

```
cp 0x03_char.uf2 /Volumes/RPI-RP2
```

Then we need to locate the USB drive so you can do the following.

```
ls /dev/tty.
```

Press tab to find the drive and then in my case I will use **screen** to connect.

```
screen /dev/tty.usbmodem0000000000001
```

You should see a an "x" being printed every second.

```
x
x
x
x
x
x
```

Next lesson we will debug char.

Part 6 - Debugging char

Today we debug the char program. Let's review the code.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    char x = 'x';

    printf("%c\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up our debugger.

```
radare2 -w arm -b 16 0x03_char.elf
```

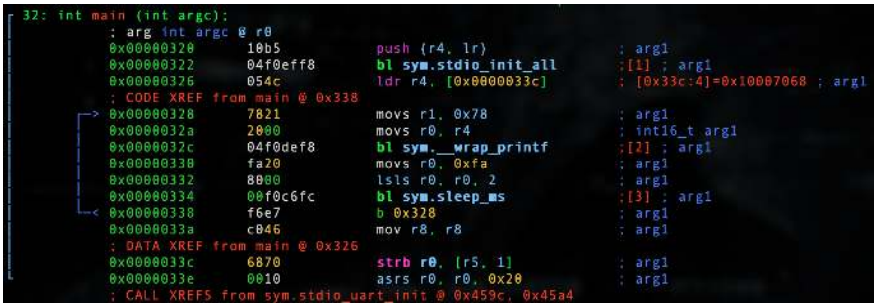
Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



We start out by setting up our main return value.

```
push {r4, lr}
```

We call the standard I/O init.

```
bl sym.stdio_init_all
```

We then load our format modifier %c into *r4*.

```
ldr r4, [0x0000033c]
```

We can prove it.

```
:> psz @ [0x0000033c]
%c
```

We then load our char *'x'* into *r1*.

```
movs r1, 0x78
```

https://www.asciitable.com

You can check with above site that 0x78 hex is *'x'*.

We then move our format modifier into *r0*.

```
movs r0, r4
```

We then branch long to the printf wrapper and call it.

```
bl sym.__wrap_printf
```

We then move 250 decimal or 0xfa hex into *r0*.

```
movs r0, 0xfa
```

We then move 250 decimal, which we know when logical shift left twice will be 1,000 decimal or 0xfa hex into *r0*.

```
lsls r0, r0, 2
```

We then call the sleep_ms function.

```
bl sym.sleep_ms
```

We then continue the while loop infinitely.

```
b 0x328
```

In our next lesson we will hack the char data type.

Part 7 - Hacking char

Today we hack the simple char program.

Let's review our code.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    char x = 'x';

    printf("%c\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up our debugger.

```
radare2 -w arm -b 16 0x03_char.elf
```

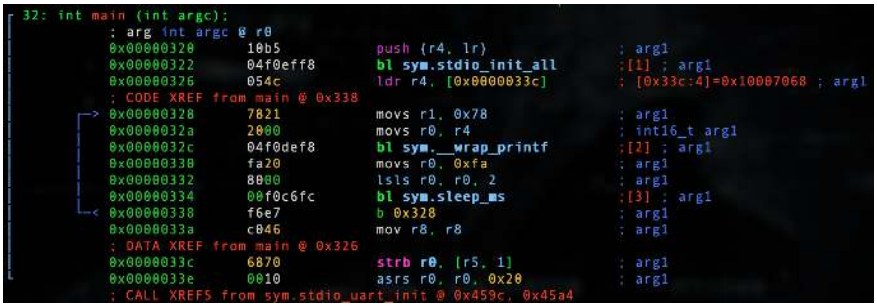
Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



In our last lesson we broke down each line. Here we are clearly interested in hacking the value of 0x78 and changing that to anything we want. Let's try 0x79. This simple hack will turn the char *'x'* into *'y'*.

```
:> wa movs r1, 0x79 @ 0x00000328
Written 2 byte(s) (movs r1, 0x79) = wx 7921
```

Let's verify the change.

```
:> pd 1 @ 0x00000328
|           ; CODE XREF from main @ 0x338
|           0x00000328      7921           movs r1, 0x79
              ; 'y' ; arg1
```

In this case our debugger is even telling us it is in fact *'y'* in addition to now we are moving the hex ascii value into 0x79 into *r1*.

Let's also hack the sleep time to 2000 ms or 2 seconds.

```
:> wa lsls r0, r0, 3 @ 0x00000332
Written 2 byte(s) (lsls r0, r0, 3) = wx c000
```

Here we simply logical shift left 3 times therefore 250 x 2 = 500, 500 x 2 = 1000, 1000 x 2 = 2000.

Let's verify.

```
:> pd 1 @ 0x00000332
|           0x00000332      c000           lsls r0, r0, 3
```

All we have to do now is exit and convert our **.elf** to **.uf2**!

```
./elf2uf2/elf2uf2 0x03_char.elf 0x03_char.uf2
```

Plug in the Pico and make sure you hold down BOOTSEL or use the setup I provided in the part 2.

```
cp 0x03_char.uf2 /Volumes/RPI-RP2
```

Let's screen it!

```
screen /dev/tty.usbmodem0000000000001
```

AHH yea!

```
y
y
y
y
y
y
```

We see 'y' printed out every 2 seconds!

In our next lesson we will discuss the int data type.

Part 8 - int

Today we are going to work with the int data type which are nothing more than whole numbers. They can be signed or unsigned as well.

Let's work with a simple example. **0x04_int.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    int x = 40;

    printf("%d\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Here we simply have our standard IO function followed by our infinite loop. We simply assign *40* to the int data type *x* and print it using the *%d* format modifier and sleep for *1* second.

Let's make a new dir **0x04_int** and add our **CMakeLists.txt** file in it.

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(0x04_int
  0x04_int.c
)

pico_enable_stdio_usb(0x04_int 1)

pico_add_extra_outputs(0x04_int)

target_link_libraries(0x04_int pico_stdlib)
```

Next we need to copy the **pico_sdk_import.cmake** file from the external folder in the **pico-sdk** installation to the **0x04_int** project folder.

```
cp ../pico-sdk/external/pico_sdk_import.cmake .
```

Finally we are ready to build.

```
mkdir build
cd build
export PICO_SDK_PATH=../../pico-sdk
cmake ..
make
```

Then simply copy the **.uf2** file to the drive.

```
cp 0x04_int.uf2 /Volumes/RPI-RP2
```

Then we need to locate the USB drive so you can do the following.

```
ls /dev/tty.
```

Press tab to find the drive and then in my case I will use **screen** to connect.

```
screen /dev/tty.usbmodem0000000000001
```

You should see a an *40* being printed every second.

```
40
40
40
40
40
40
40
40
40
40
40
40
```

In our next lesson we will debug.

Part 9 - Debugging int

Today we are going to debug our very simple int program. Let's review the code.

0x04_int.c

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    int x = 40;

    printf("%d\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up in our debugger.

```
radare2 -w arm -b 16 0x04_int.elf
```

Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



We start out by setting up our main return value.

```
push {r4, lr}
```

We call the standard I/O init.

```
bl sym.stdio_init_all
```

We then load our format modifier %d into *r4*.

```
ldr r4, [0x0000033c]
```

We can prove it.

```
:> psz @ [0x0000033c]
%d
```

We then load our int *'40'* into *r1 _which is _0x28* hex.

```
movs r1, 0x28
```

We can prove it.

```
:> ? 0x28
int32   40
uint32  40
hex     0x28
octal   050
unit    40
segment 0000:0028
string  "("
fvalue: 40.0
float:  0.000000f
double: 0.000000
binary  0b00101000
ternary 0t1111
```

We then move our format modifier into *r0*.

```
movs r0, r4
```

We then branch long to the printf wrapper and call it.

```
bl sym.__wrap_printf
```

We then move 250 decimal or 0xfa hex into *r0*.

```
movs r0, 0xfa
```

We then move 250 decimal, which we know when logical shift left twice will be 1,000 decimal or 0xfa hex into *r0*.

```
lsls r0, r0, 2
```

We then call the sleep_ms function.

```
bl sym.sleep_ms
```

We then continue the while loop infinitely.

```
b 0x328
```

In our next lesson we will hack this very simple binary.

Part 10 - Hacking int

Today we hack our simple int program. Let's review the code.

0x04_int.c

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    int x = 40;

    printf("%d\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up in our debugger.

```
radare2 -w arm -b 16 0x04_int.elf
```

Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



We are going to first hack the int value which we know is *40* decimal or *28* hex.

```
:> wa movs r1, 0x30 @ 0x00000328
Written 2 byte(s) (movs r1, 0x30) = wx 3021
```

Here we see *0x30* is *48* decimal.

```
:> ? 0x30
int32   48
uint32  48
hex     0x30
octal   060
unit    48
segment 0000:0030
string  "0"
fvalue: 48.0
float:  0.000000f
double: 0.000000
binary  0b00110000
ternary 0t1210
```

We also see that *0xfa* which we know is *250* decimal is our 1/4 millisecond delay that when shifted left twice, multiplies, and becomes *1000* decimal for 1 second delay.

```
:> ? 0xfa
int32   250
uint32  250
hex     0xfa
octal   0372
unit    250
segment 0000:00fa
string  "\xfa"
fvalue: 250.0
float:  0.000000f
double: 0.000000
binary  0b11111010
ternary 0t100021
```

Let's hack that to *50* decimal.

```
:> wa movs r0, 0x32 @ 0x00000330
Written 2 byte(s) (movs r0, 0x32) = wx 3220
```

We can see that it is in fact *50* decimal.

```
:> ? 0x32
int32   50
uint32  50
hex     0x32
octal   062
unit    50
segment 0000:0032
string  "2"
fvalue: 50.0
float:  0.000000f
double: 0.000000
binary  0b00110010
ternary 0t1212
```

Let's also only shift it left once such that it will take *50* decimal and turn it into *100* when it shifts left only once.

```
:> wa lsls r0, r0, 1 @ 0x00000332
Written 2 byte(s) (lsls r0, r0, 1) = wx 4000
```

All we have to do now is exit and convert our **.elf** to **.uf2**!

```
./elf2uf2/elf2uf2 0x04_int.elf 0x04_int.uf2
```

Plug in the Pico and make sure you hold down BOOTSEL or use the setup I provided in the part 2.

```
cp 0x04_int.uf2 /Volumes/RPI-RP2
```

Let's screen it!

```
screen /dev/tty.usbmodem0000000000001
```

AHH yea!

```
48
48
48
48
48
48
48
48
48
48
48
48
48
48
48
48
48
48
48
```

Here we see we hacked it to 48 decimal and it is printing every 100 milliseconds!

In our next lesson we will deal with floats and the unique way the Pico handles them as it does not have a co-processor.

Part 11 - float

Today we are going to handle the float data type. In the Pico there is no co-processor to handle floating-point numbers as this is handled through a series of functionality through software in the API.

Let's work with a simple example. **0x05_float.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    float x = 40.5;

    printf("%f\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Very simply we assign a float of *40.5* into *x* and print it with the *%f _format modifier and then sleep for _1* second.

Let's make a new dir **0x05_float** and add our **CMakeLists.txt** file in it.

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(0x05_float
  0x05_float.c
)

pico_enable_stdio_usb(0x05_float 1)

pico_add_extra_outputs(0x05_float)

target_link_libraries(0x05_float pico_stdlib)
```

Next we need to copy the **pico_sdk_import.cmake** file from the external folder in the **pico-sdk** installation to the **0x05_float** project folder.

```
cp ../pico-sdk/external/pico_sdk_import.cmake .
```

Finally we are ready to build.

```
mkdir build
cd build
export PICO_SDK_PATH=../../pico-sdk
cmake ..
make
```

Then simply copy the **.uf2** file to the drive.

```
cp 0x05_float.uf2 /Volumes/RPI-RP2
```

Then we need to locate the USB drive so you can do the following.

```
ls /dev/tty.
```

Press tab to find the drive and then in my case I will use **screen** to connect.

```
screen /dev/tty.usbmodem0000000000001
```

You should see a an *40.5* being printed every second.

```
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
```

In our next lesson we will debug.

Part 12 - Debugging float

Let's review our example. **0x05_float.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    float x = 40.5;

    printf("%f\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up in our debugger.

```
radare2 -w arm -b 16 0x05_float.elf
```

Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



We see the format specifier in *[0x0000033c].*

```
:> psz @ [0x0000033c]
%f
```

The float is at *[0x00000340]*.

```
:> pff @ [0x00000340]
0x00004000 = 9.32830524e-09
```

Do not worry that the float is inaccurate as this machine is x64. What is important to see is the value *0x00004000*. You then ask yourself, hey, that is not *40.5*! What is the deal?

OK...

The Pico does not have its own math coprocessor so it handles floats and doubles using software. Therefore *0x00004000* would be the representation of *40.5* decimal.

So if the value was *40.4*, for example, it would be *0x00003333*. Conversely *40.6* would be *0x00004ccc*.

Take a look at the following table which will help illustrate the point.

```
0x3ff00000 = 1.000000
0x3ff00001 = 1.000001
0x3ff00002 = 1.000002
...
0x3ff0000f = 1.000015
0x3ff00010 = 1.000016
0x3ff00011 = 1.000017
etc...
```

Ultimately the values in these 4 bytes (32-bits) will determine the value of the float.

In our next lesson we will hack the float and demonstrate this logic.

Part 13 - Hacking float

Let's review our example. **0x05_float.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    float x = 40.5;

    printf("%f\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up in our debugger.

```
radare2 -w arm -b 16 0x05_float.elf
```

Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



The float is at *[0x00000340]*.

```
:> pff @ [0x00000340]
0x00004000 = 9.32830524e-09
```

As we discussed in the last lesson, do not worry that the float is inaccurate as this machine is x64. What is important to see is the value *0x00004000*.

In our last lesson we also explained the way the Pico handles floats. Let's review some basics.

```
0x3ff00000 = 1.000000
0x3ff00001 = 1.000001
0x3ff00002 = 1.000002
...
0x3ff0000f = 1.000015
0x3ff00010 = 1.000016
0x3ff00011 = 1.000017
etc...
```

Let's hack to 1.000000 as follows.

Our microcontroller is a little endian architecture therefore if we are going to change our 40.5 to 1.0 we need to put that value in reverse byte order therefore...

```
0x3ff00000
```

Needs to be...

```
0x0000f03f
```

Therefore we need to change the value at the following.

```
wx 0x0000f03f @ 0x00000340
```

All we have to do now is exit and convert our **.elf** to **.uf2**!

```
./elf2uf2/elf2uf2 0x05_float.elf 0x05_float.uf2
```

Plug in the Pico and make sure you hold down BOOTSEL or use the setup I provided in the part 2.

```
cp 0x05_float.uf2 /Volumes/RPI-RP2
```

Let's screen it!

```
screen /dev/tty.usbmodem0000000000001
```

AHH yea!

```
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
1.000000
```

Here we have hacked the value to 1.000000 and we let the 1 second sleep to persist.

In our next lesson we will discuss the double data type.

Part 14 - double

Today we are going to handle the double data type. As we discussed, in the Pico there is no co-processor to handle floating-point numbers as this is handled through a series of functionality through software in the API. This is the same with double-precision.

Let's work with a simple example. **0x06_double.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    double x = 40.5;

    printf("%f\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Very simply we assign a float of *40.5* into *x* and print it with the *%f _format modifier and then sleep for _1* second.

Let's make a new dir **0x06_double** and add our **CMakeLists.txt** file in it.

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(0x06_double
  0x06_double.c
)

pico_enable_stdio_usb(0x06_double 1)

pico_add_extra_outputs(0x056_double)

target_link_libraries(0x06_double pico_stdlib)
```

Next we need to copy the **pico_sdk_import.cmake** file from the external folder in the **pico-sdk** installation to the **0x06_double** project folder.

```
cp ../pico-sdk/external/pico_sdk_import.cmake .
```

Finally we are ready to build.

```
mkdir build
cd build
export PICO_SDK_PATH=../../pico-sdk
cmake ..
make
```

Then simply copy the **.uf2** file to the drive.

```
cp 0x06_double.uf2 /Volumes/RPI-RP2
```

Then we need to locate the USB drive so you can do the following.

```
ls /dev/tty.
```

Press tab to find the drive and then in my case I will use **screen** to connect.

```
screen /dev/tty.usbmodem0000000000001
```

You should see a an *40.5* being printed every second.

```
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
40.500000
```

In our next lesson we will debug.

Part 15 - Debugging double

Let's review **0x06_double.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    double x = 40.5;

    printf("%f\n", x);

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up in our debugger.

```
radare2 -w arm -b 16 0x06_double.elf
```

Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



We see the format specifier in *[0x0000033c].*

```
:> psz @ [0x0000033c]
%f
```

The double is at *[0x00000340]*.

```
:> pff @ [0x00000340]
0x00004000 = 9.32830524e-09
```

Ok... Same deal as the float lesson so why did I waste time on choosing 40.5?

I wanted to show you definitive proof that the compiler will treat this the same as it is within the bounds of a float when the Pico SDK functionality does it's magic as there is NO co-processor.

Let's examine a MOD to our program.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    double x = 40.55555555555555555;

    printf("%.16f\n", x)

    sleep_ms(1000);
  }

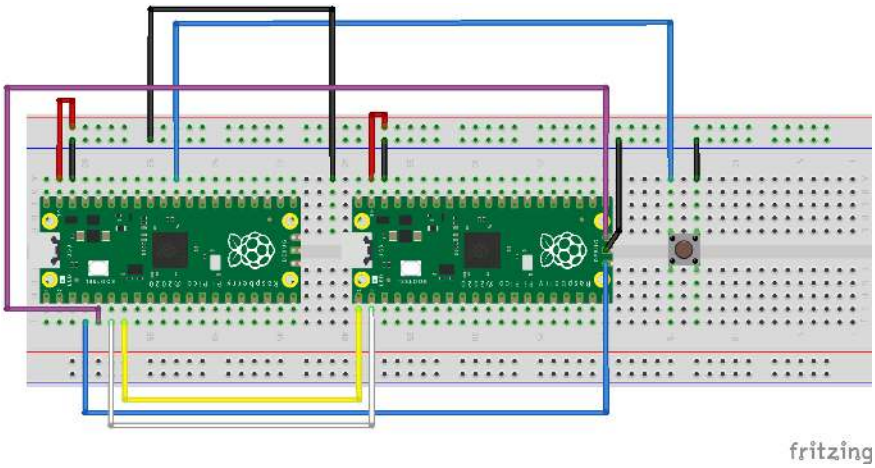
  return 0;
}
```

When we compile and run this program we get the following.

```
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
```

OK well... This looks different. Let us for the first time in this course look at a Dynamic Reverse Engineering analysis in GDB.

It is NOT critical here that you run this and set this all up in GDB as there are a great deal of steps in addition to another Pico needed in a configuration such as the following below.



The scope of this course is to understand Static Reverse Engineering however I wanted to depart and show you what GDB is showing us with this new binary.

It is NOT necessary to use Dynamic Reverse Engineering unless you are dealing with a situation where you have a packed binary that you have to dynamically load and write out the code. It does make things easier when you are using Dynamic Reverse Engineering however I want to show you that Static Reverse Engineering can get you everything you need without having to set up a remote process to actually run the binary on.

If you did find it necessary to try this you would need to first install the OpenOCD repo into the pico folder that we created at the beginning of this course. You can find details at the link below and go to **5.1 Installing OpenOCD** in the datasheet.

https://datasheets.raspberrypi.org/pico/getting-started-with-pico.pdf

You will then need to visit the page below and download the uf2 located at **Debugging using another Raspberry Pi Pico** and then flash the first Pico with the uf2.

https://www.raspberrypi.org/documentation/rp2040/getting-started/\#board-specifications

TERMINAL 1: You will then need to set up a first terminal to go into the openocd folder and run the following.

```
src/openocd -f interface/picoprobe.cfg -f
target/rp2040.cfg -s tcl
```

TERMINAL 2: You will need to go into the build folder of your project and run the following.

```
arm-none-eabi-gdb 0x06_double.elf
target extended-remote localhost:3333
load
monitor reset init
b main
c
```

TERMINAL 3: You will need to run the screen emulator which will start with a blinking cursor.

```
screen /dev/tty.usbmodem14101 115200
```

Nonetheless with that brief explanation, lets' review this dynamically in GDB.

```
Dump of assembler code for function main:
   0x10000320 <+0>:     push    {r4, lr}
=> 0x10000322 <+2>:     bl      0x1000450c <stdio_init_all>
   0x10000326 <+6>:     ldr     r4, [pc, #20]    ; (0x1000033c <main+28>)
   0x10000328 <+8>:     movs    r0, r4
   0x1000032a <+10>:    ldr     r2, [pc, #20]    ; (0x10000340 <main+32>)
   0x1000032c <+12>:    ldr     r3, [pc, #20]    ; (0x10000344 <main+36>)
   0x1000032e <+14>:    bl      0x100044f4 <__wrap_printf>
   0x10000332 <+18>:    movs    r0, #250         ; 0xfa
   0x10000334 <+20>:    lsls    r0, r0, #2
   0x10000336 <+22>:    bl      0x10000ccc <sleep_ms>
   0x1000033a <+26>:    b.n     0x10000328 <main+8>
   0x1000033c <+28>:    strb    r0, [r6, #1]
   0x1000033e <+30>:    asrs    r0, r0, #32
   0x10000340 <+32>:    adds    r2, r6, #1
   0x10000342 <+34>:    strb    r7, [r0, #7]
   0x10000344 <+36>:    bxns    r3
   0x10000346 <+38>:    eors    r4, r0
End of assembler dump.
```

We see two values at *0x10000340* and *0x10000344*.

Let's delete all breakpoints and break right before the call to the printf wrapper.

```
d
b *0x1000032e
c
```

Let's examine the values at each of these locations.

```
p/x *0x10000340
0x71c71c72

p/x *0x10000344
0x4044471c
```

We know that the following output is what prints.

```
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
40.5555555560000000
```

What is happening is that these values are now in R2 and R3 respectively.

```
p/x $r2
0x71c71c72

p/x $r3
0x4044471c
```

In ARM 32 Assembly the arguments to the functions are passed in r0-r3 and if you need more args they are put on the stack. In our case r0 has our format modifier.

```
x/s $r0
0x10007070:    "%.16f\n"
```

We see in r1 a value pointing to the stack.

```
x/w $r1
0x0:    0x20041f00

p/x *0x20041f00
0xa
```

This is another piece going into the printf wrapper in order to properly print the string to the STDOUT.

In our next lesson we will hack statically.

Part 16 - Hacking double

Let's review **0x06_double_MOD.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"

int main()
{
  stdio_init_all();

  while(1)
  {
    double x = 40.55555555555555555;

    printf("%.16f\n", x)

    sleep_ms(1000);
  }

  return 0;
}
```

Let's fire up in our debugger.

```
radare2 -w arm -b 16 0x06_double.elf
```

Let's auto analyze.

```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.



Our microcontroller is a little endian architecture as we have discussed before therefore if we are going to change our 40.5555555560000000 to 1.0 we need to put that value in reverse byte order therefore...

```
0x3ff00000
```

Needs to be...

```
0x0000f03f
```

Therefore we need to change the value at the following.

```
wx 0x0000f03f @ 0x00000344
```

All we have to do now is exit and convert our **.elf** to **.uf2**!

```
./elf2uf2/elf2uf2 0x06_double.elf 0x06_double.uf2
```

Plug in the Pico and make sure you hold down BOOTSEL or use the setup I provided in the part 2.

```
cp 0x06_double.uf2 /Volumes/RPI-RP2
```

Let's screen it!

```
screen /dev/tty.usbmodem0000000000001
```

AHH yea!

```
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
1.0000002380000000
```

Now we should have a good understanding of the data types within C to look at some slightly larger concepts.

In our next lesson we will begin to discuss input.

Part 17 - "ABSOLUTE POWER CORRUPTS ABSOLUTELY!", The Tragic Tale Of Input...

"But I am just here to learn Reverse Engineering I am really not interested in the non-sexy coding part, I just want the Reverse Engineering challenge and be a superstar!"

Ahh the naivety of the non-Jedi. For much they have to learn or perhaps unlearn to really learn!

I take not a shot at programming books and courses that teach how to capture STDIN from users in a simplistic manner like 'scanf' however I rather challenge YOU to consider a proper approach.

We are dealing with a microcontroller. It is THE target of Ransomware Authors, State Agents and all sorts of unsavory parties. WE must FIRST take TIME to understand how to properly handle input regarding a microcontroller.

I have taken the liberty to construct a proper input function for your examination.

```
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"

#define ZERO 0x30
#define NINE 0x39
#define PERIOD 0x2e
#define CAPITAL_A 0x41
#define LOWER_CASE_Z 0x7a
#define BACKSPACE 0x08
#define DEL 0x7f

void input_proc(char type, char* p_usb_char, char*
p_usb_string, const int* p_USB_STRING_SIZE)
{
  *p_usb_char = '\0';
  *p_usb_char = getchar_timeout_us(0);
  if(*p_usb_char == BACKSPACE || *p_usb_char == DEL)
  {
    if(p_usb_string[0] != '\0')
    {
      printf("\b");
      printf(" ");
      printf("\b");
      p_usb_string[strlen(p_usb_string)-1] = '\0';
    }
  }
  if(type == 'f')
  {
    char* period;
    while((*p_usb_char >= ZERO && *p_usb_char <= NINE) ||
*p_usb_char == PERIOD)
    {
      if(*p_usb_char == PERIOD)
        period = strchr(p_usb_string, '.');
      if(period == NULL)
      {
        if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
        {
          putchar(*p_usb_char);
          strncat(p_usb_string, p_usb_char, 1);
        }
        *p_usb_char = '\0';
      }
      else
        break;
    }
  }
```

```
  else if(type == 'd')
  {
    while(*p_usb_char >= ZERO && *p_usb_char <= NINE)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
  else if(type == 's')
  {
    while(*p_usb_char >= CAPITAL_A && *p_usb_char <=
LOWER_CASE_Z)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
}
```

"Woah I thought we were taking it slow!" The time has come to properly start to understand how to be a Jedi when designing effective software. The TIME has come to take the time to properly digest a REAL input validation function.

I want you to take the time and digest this function so that we can review it in the next lesson.

In our next lesson we will properly break down this work of genius to properly understand and craft and ultimately Reverse Engineer in our coming future!

Part 18 - "FOR 800 YEARS HAVE I TRAINED JEDI!", The FORCE That IS Input...

"The year is 2021 and seven months, the average price of a gallon of gas within the United States is $7.51 a gallon. Four other U.S. Pipelines were compromised with Ransomware and the Five Eyes discovered a compromised network within one of the water supplies within a major metropolitan U.S. City."

"Intelligence sources have located the HQ of the 'Dark Eyes' organization behind the malware attacks and utilize a Pico Microcontroller as the controller inside a drone which is gearing up to strike this facility and knock out their communications to avoid the attack on our water supply."

"The attack coordinates are '61.013693050912785, 99.19670587477269' to which the Drone Operator enters in, '61.013693050912785, 9e.19670587477269', which is 'Mir Mines, Russia'. They launch the drone and it detonates at, '61.013693050912785, 9.19670587477269', which is 'Nord-Aurdal Municipality, Norway'."

"Panic ensues however DHS was able to secure the water supply network before Ransomware was able to encrypt their network and within twelve hours the network was fully secured."

Ok...

I wanted to take the time to really show the absolute CRITICALITY of designing software with proper input handling. Using 'scanf' or other techniques which do not properly handle every keystroke can lead to a situation like the one outlined above.

Let's review our input function...

```
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"

#define ZERO 0x30
#define NINE 0x39
#define PERIOD 0x2e
#define CAPITAL_A 0x41
#define LOWER_CASE_Z 0x7a
#define BACKSPACE 0x08
#define DEL 0x7f

void input_proc(char type, char* p_usb_char, char*
p_usb_string, const int* p_USB_STRING_SIZE)
{
  *p_usb_char = '\0';
  *p_usb_char = getchar_timeout_us(0);
  if(*p_usb_char == BACKSPACE || *p_usb_char == DEL)
  {
    if(p_usb_string[0] != '\0')
    {
      printf("\b");
      printf(" ");
      printf("\b");
      p_usb_string[strlen(p_usb_string)-1] = '\0';
    }
  }
  if(type == 'f')
  {
    char* period;
    while((*p_usb_char >= ZERO && *p_usb_char <= NINE) ||
*p_usb_char == PERIOD)
    {
      if(*p_usb_char == PERIOD)
        period = strchr(p_usb_string, '.');
      if(period == NULL)
      {
        if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
        {
          putchar(*p_usb_char);
          strncat(p_usb_string, p_usb_char, 1);
        }
        *p_usb_char = '\0';
      }
      else
        break;
    }
  }
```

```
  else if(type == 'd')
  {
    while(*p_usb_char >= ZERO && *p_usb_char <= NINE)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
  else if(type == 's')
  {
    while(*p_usb_char >= CAPITAL_A && *p_usb_char <=
LOWER_CASE_Z)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
}
```

Today we are going to go over exactly what this function is actually doing.

```
void input_proc(char type, char* p_usb_char, char*
p_usb_string, const int* p_USB_STRING_SIZE)
```

We begin with the function header. We first are taking a *char* of *type* where in our example we will use *'f'* for handling floating-point numbers. We then have a *char** (pointer) *p_usb_char* which will be init to *'\0'* in **main.c**. We then have a char* p_usb_string which we will be init to *'\0'* in **main.c**. We then have a *const int** *p_USB_STRING_SIZE* which will be init to *100* in **main.c**.

We then create logic to properly handle a delete or backspace button.

```
  if(*p_usb_char == BACKSPACE || *p_usb_char == DEL)
  {
    if(p_usb_string[0] != '\0')
    {
      printf("\b");
      printf(" ");
      printf("\b");
      p_usb_string[strlen(p_usb_string)-1] = '\0';
    }
  }
```

We then create logic to handle if the main.c program is expecting ONLY floating-point numbers as in our story above if would have been implemented the drone would not have missed their target.

```
  if(type == 'f')
  {
    char* period;
    while((*p_usb_char >= ZERO && *p_usb_char <= NINE) ||
*p_usb_char == PERIOD)
    {
      if(*p_usb_char == PERIOD)
        period = strchr(p_usb_string, '.');
      if(period == NULL)
      {
        if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
        {
          putchar(*p_usb_char);
          strncat(p_usb_string, p_usb_char, 1);
        }
        *p_usb_char = '\0';
      }
      else
        break;
    }
  }
```

We see that if someone enters anything other than a *ZERO* through *NINE* or a *PERIOD,* the input will SIMPLY BE REJECTED!

You also see that if there is a *PERIOD* entered a second one could not be entered either maliciously or by accident. We also handle the amount of input to be less than *100* properly. We then properly build our string from every properly cleaned keystroke.

Similar logic handles if you are dealing with decimals or strings.

```
  else if(type == 'd')
  {
    while(*p_usb_char >= ZERO && *p_usb_char <= NINE)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
  else if(type == 's')
  {
    while(*p_usb_char >= CAPITAL_A && *p_usb_char <=
LOWER_CASE_Z)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
```

In our next lesson we will implement this in our Pico microcontroller.

Part 19 - Input

The last two lessons hopefully showcased the need for a mature approach to handling input on any serious application.

Today we will design a proper input architecture for the Pico related to STDIN and STDIO.

Let's begin with creating an **input.h** as follows.

```
void input_proc(char type, char* p_usb_char, char*
p_usb_string, const int* p_USB_STRING_SIZE);
void flush_input(char* p_usb_string);
```

Here we setup our input header file to address the params that we discussed in the last lesson. We also set up our *flush_input* function to handle clearing the input buffer after it is used to ensure it is clean before new input is obtained for another call to *input_proc*.

Next we will create our **print.h** as follows.

```
void print_proc(char* p_usb_char, char* p_usb_string);
```

Very simply we are going to pass in a char array from the caller to handle each char and a char array from the caller to handle the string creation.

Next we will create our **input.c** as follows.

```
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"

#define ZERO 0x30
#define NINE 0x39
#define PERIOD 0x2e
#define CAPITAL_A 0x41
#define LOWER_CASE_Z 0x7a
#define BACKSPACE 0x08
#define DEL 0x7f

void input_proc(char type, char* p_usb_char, char*
p_usb_string, const int* p_USB_STRING_SIZE)
{
  *p_usb_char = '\0';
  *p_usb_char = getchar_timeout_us(0);
  if(*p_usb_char == BACKSPACE || *p_usb_char == DEL)
  {
    if(p_usb_string[0] != '\0')
    {
      printf("\b");
      printf(" ");
      printf("\b");
      p_usb_string[strlen(p_usb_string)-1] = '\0';
    }
  }
  if(type == 'f')
  {
    char* period;
    while((*p_usb_char >= ZERO && *p_usb_char <= NINE) ||
*p_usb_char == PERIOD)
    {
      if(*p_usb_char == PERIOD)
        period = strchr(p_usb_string, '.');
      if(period == NULL)
      {
        if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
        {
          putchar(*p_usb_char);
          strncat(p_usb_string, p_usb_char, 1);
        }
        *p_usb_char = '\0';
      }
      else
        break;
    }
  }
```

```
  else if(type == 'd')
  {
    while(*p_usb_char >= ZERO && *p_usb_char <= NINE)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
  else if(type == 's')
  {
    while(*p_usb_char >= CAPITAL_A && *p_usb_char <=
LOWER_CASE_Z)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
}

void flush_input(char* p_usb_string)
{
  p_usb_string[0] = '\0';
}
```

Everything should be fully understood at this point with the above. If it is not please review the last two lessons.

Next we will create our **print.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "input.h"

#define RETURN 0x0d

void print_proc(char* p_usb_char, char* p_usb_string)
{
  if(*p_usb_char == RETURN)
  {
    if(p_usb_string[0] == '\0')
      printf("\n");
    else
      printf("\n%s\n", p_usb_string);
    flush_input(p_usb_string);
  }
}
```

Here we bring in our char and string capability and if the return key is pressed will print the contents of the string and then call the *flush_input* to clear the buffer as discussed.

Finally we will create our **main.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "print.h"
#include "input.h"

int main()
{
  stdio_init_all();

  const int USB_STRING_SIZE = 100;
  char usb_char;
  usb_char = '\0';
  char usb_string[USB_STRING_SIZE];
  usb_string[0] = '\0';

  while(1)
  {
    input_proc('f', &usb_char, usb_string,
&USB_STRING_SIZE);
    print_proc(&usb_char, usb_string);
  }

  return 0;
}
```

Here we simply set up our input procedure to handle float input.

Let's make a new dir **0x07_input** and add our **CMakeLists.txt** file in it.

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_C_FLAGS_RELEASE "${CMAKE_C_FLAGS_RELEASE}")
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE}")
pico_sdk_init()

add_executable(main
  main.c
  print.c
  input.c
)

pico_enable_stdio_usb(main 1)
pico_enable_stdio_uart(main 0)
pico_add_extra_outputs(main)

target_link_libraries(main pico_stdlib hardware_i2c)

add_custom_target(flash
    COMMAND cp main.uf2 /Volumes/RPI-RP2/
    DEPENDS main
)
```

Next we need to copy the **pico_sdk_import.cmake** file from the external folder in the **pico-sdk** installation to the **0x07_input** project folder.

```
cp ../pico-sdk/external/pico_sdk_import.cmake .
```

Finally we are ready to build.

```
mkdir build
cd build
export PICO_SDK_PATH=../../pico-sdk
cmake ..
make
make flash
```

I added a flash routine in the makefile to save us time from copying to the Pico. Remember to put the Pico into flash mode first.

Then we need to locate the USB drive so you can do the following.

```
ls /dev/tty.
```

Press tab to find the drive and then in my case I will use **screen** to connect.

```
screen /dev/tty.usbmodem0000000000001
```

Boom! Now you will see you will ONLY be able to enter in numbers and ONLY ONE decimal point. We properly handle for backspacing and when you reach the max of 100 chars it will not allow you to type further. Finally it prints back what you typed.

```
32.3333
32.3333
32.11111111
32.11111111
7.99999003902930420384802384082304820384028342340284923840238948230482938429034823948293849023849223
7.99999003902930420384802384082304820384028342340284923840238948230482938429034823948293849023849223
```

In our next lesson we will debug.

Part 20 - Debugging Input

Today we will debug our input function. Let's review our code.

Review **input.c** as follows.

```
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"

#define ZERO 0x30
#define NINE 0x39
#define PERIOD 0x2e
#define CAPITAL_A 0x41
#define LOWER_CASE_Z 0x7a
#define BACKSPACE 0x08
#define DEL 0x7f

void input_proc(char type, char* p_usb_char, char*
p_usb_string, const int* p_USB_STRING_SIZE)
{
  *p_usb_char = '\0';
  *p_usb_char = getchar_timeout_us(0);
  if(*p_usb_char == BACKSPACE || *p_usb_char == DEL)
  {
    if(p_usb_string[0] != '\0')
    {
      printf("\b");
      printf(" ");
      printf("\b");
      p_usb_string[strlen(p_usb_string)-1] = '\0';
    }
  }
  if(type == 'f')
  {
    char* period;
    while((*p_usb_char >= ZERO && *p_usb_char <= NINE) ||
*p_usb_char == PERIOD)
    {
      if(*p_usb_char == PERIOD)
        period = strchr(p_usb_string, '.');
      if(period == NULL)
      {
        if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
        {
          putchar(*p_usb_char);
          strncat(p_usb_string, p_usb_char, 1);
        }
        *p_usb_char = '\0';
      }
      else
        break;
    }
  }
```

```
  else if(type == 'd')
  {
    while(*p_usb_char >= ZERO && *p_usb_char <= NINE)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
  else if(type == 's')
  {
    while(*p_usb_char >= CAPITAL_A && *p_usb_char <=
LOWER_CASE_Z)
    {
      if(strlen(p_usb_string) < *p_USB_STRING_SIZE)
      {
        putchar(*p_usb_char);
        strncat(p_usb_string, p_usb_char, 1);
      }
      *p_usb_char = '\0';
    }
  }
}

void flush_input(char* p_usb_string)
{
  p_usb_string[0] = '\0';
}
```

Review our **print.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "input.h"

#define RETURN 0x0d

void print_proc(char* p_usb_char, char* p_usb_string)
{
  if(*p_usb_char == RETURN)
  {
    if(p_usb_string[0] == '\0')
      printf("\n");
    else
      printf("\n%s\n", p_usb_string);
    flush_input(p_usb_string);
  }
}
```

Review our **main.c** as follows.

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "print.h"
#include "input.h"

int main()
{
  stdio_init_all();

  const int USB_STRING_SIZE = 100;
  char usb_char;
  usb_char = '\0';
  char usb_string[USB_STRING_SIZE];
  usb_string[0] = '\0';

  while(1)
  {
    input_proc('f', &usb_char, usb_string,
&USB_STRING_SIZE);
    print_proc(&usb_char, usb_string);
  }

  return 0;
}
```

Let's fire up in our debugger.

```
radare2 -w arm -b 16 main.elf
```

Let's auto analyze.

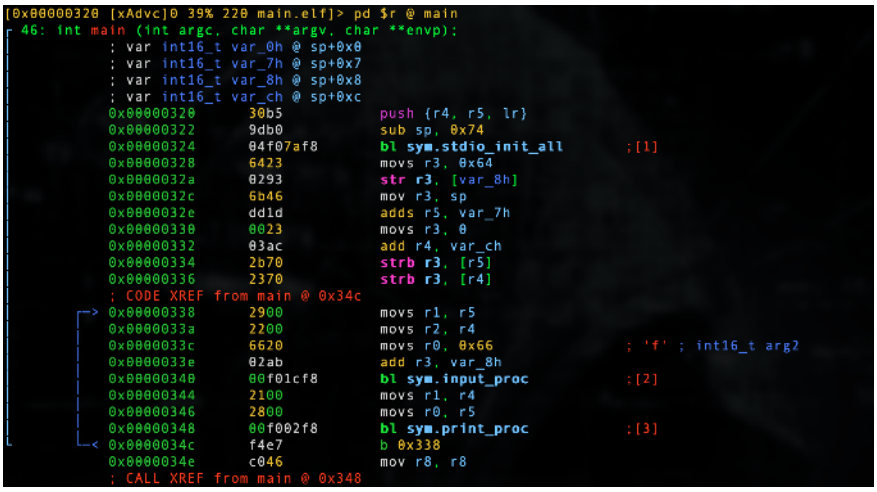
```
aaaa
```

Let's seek to main.

```
s main
```

Let's go into visual mode by typing **V** and then **p** twice to get to a good debugger view.

We first review *main*.



We see our *stdio_init_all* call which sets up IO and we see a *0x64* into *r3* which is our move of 100 decimal to set *USB_STRING_SIZE _and we set up our _usb_char* value and init to *0* and finally *usb_string* and init to *0*.

Let's look at our *print_proc* function.



We first check to see if our pointer to usb_char or *p_usb_char* is equal to the *RETURN* key or *0xd* and if so branch.

We then iterate over *p_usb_string* until we hit the null terminator and then call our _printf _function which as we can see here is a wrapper to the c printf function.

We finally *flush_input*.

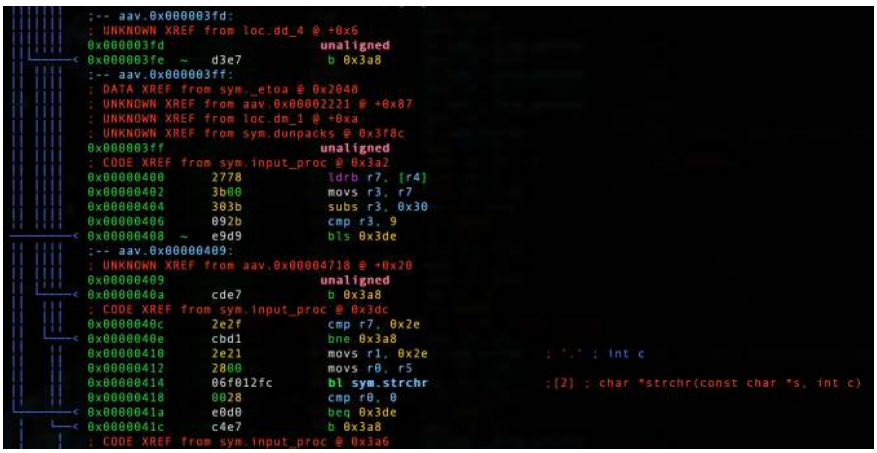
Our *input_proc* function is a bit more complex.



Here we use the g*etchar_timeout_us* function and handle the *BACKSPACE* and *DELETE* keys.



We then call our *putchar _wrapper against* 0 *and _9* and check the *strlen* and properly build our string with *strncat*.



We then properly handle our *PERIOD* logic to ensure only one _PERIOD _is entered as a floating point number can NOT handle 2 periods.



We then then properly handle our loop.

Finally, we have our *flush_input* function.



Here we simply flush the input buffer by setting *p_usb_string* to a null char.

This was a larger debug session so please take your time and compare the assembly against the source so you can really grasp each paragraph as I cover it here.

This brings us to the end of our initial learning journey. In this journey we took 197 steps together through several different architectures. It is your turn to take this training into practice and do great things!

This book will be your reference guide as you encounter challenges however there is nothing you can't accomplish!